Departamento de ingeniería informática

Trabajo fin de grado

DESARROLLO DE UNA EXTENSIÓN DE MPI PARA C++

Autor: Francisco Rodríguez Melgar
Tutor: David Expósito Singh

Madrid, junio de 2017

Agradecimientos

No tengo ninguna duda de que este trabajo y el grado al que me permite optar es el proyecto vital más importante que he realizado hasta la fecha. Es, además, la obra de afirmación personal más grande que he hecho, dado que culmina el trabajo desarrollado en 4 años de esfuerzo, colmados a veces de alegrías y a veces de esfuerzo que parecía fútil y agotador. Pero, si todo sale bien y la divina providencia así lo quiere, habré concluido con éxito esta etapa de mi vida.

Pero sería no sólo arrogante, sino también desagradecido pensar que he llegado a este punto de mi vida sólo por mis méritos o por mi sola voluntad. Como decía el filósofo español, «Yo soy yo y mi circunstancia.». Y no hay circunstancia más poderosa que la ya mencionada providencia. El primer agradecimiento que deseo hacer es, y creo que no podría ser de otro modo, a Dios Todopoderoso. Gracias a Dios por permitirme vivir hoy, por permitirme tener la suerte de haber vivido esta vida y de seguir viviéndola, y gracias por dar a la humanidad el don de la razón, del conocimiento, que es el que me ha llevado hasta el día de hoy y el que construye nuestro mundo. Hago esta mención consciente de que no nos acordamos tan a menudo como deberíamos de hacerla.

Después de saldada la debida deuda con El Creador, no puede uno pensar en otra persona que es casi tan importante como Dios para un hombre. Quiero dar las gracias a mi madre, no sólo por darme la vida, sino por volcar sus esfuerzos y desvelos en mí, de tal modo que haya podido llegar hasta aquí. Gracias por seguir ahí y por darme las herramientas que he tenido para llegar hasta aquí, por hacerme persona, sé que con una madre que no fueras tú hoy quizás no estaría aquí, no hay palabras en el mundo para calcular todo lo que has hecho por mí.

Inmediatamente detrás de mi madre viene toda mi familia: mi abuela, mis tíos y tías, mis primos… todos vosotros, que también me han ayudado se ser mejor persona cada día, a completarme y a superar mis propios defectos en muchos aspectos, que se han preocupado por mí y que han vivido conmigo los buenos y malos días de mi vida, por todo esto gracias a vosotros también, no concibo mi vida sin vosotros y por ello tampoco puedo olvidarme de vosotros en este momento.

Además, también tengo una deuda impagable con todos mis amigos, los de verdad, los mejores, aquéllos que echas de menos a los pocos días. Esos amigos en los que piensas con un cariño casi familiar. Adrián, Mireya, Eliseo, Luis, Jonathan… puede que nunca os lo haya dicho, pero siempre he creído que, si no os hubiera conocido, sería francamente una peor persona y, por ello, sé que me habéis ayudado a llegar hasta aquí.

Además, otras personas especiales que se han cruzado en mi camino y me han inspirado, me han dado experiencias nuevas o me han hecho conocer sentimientos que no conocía, pensar en temas que me eran nuevos y vivir una vida más rica, gracias a personas como Adán, como José Luis por haberos cruzado en mi camino. Siempre habéis demostrado una confianza en mí de la que yo mismo carecía, y esa manera de actuar demuestra un aprecio terriblemente genuino.

Además, un combatiente no puede olvidarse en momentos como estos de sus compañeros de batallas, de aquéllos con los que estuvo hombro con hombro y escudo con escudo, aquéllos compañeros que me han acompañado en mi viaje en esta etapa de mi vida desde que entré por primera vez en esta universidad hace 4 años ya. Jose Luis, Juan, Marga, Alejandro (los dos), Luis, Cristian y muchos más que no cabrían, muchas gracias por permitirme compartir con vosotros este tiempo. Mención especial hay que hacer a Imanol, si hablamos de batallas junto a él libré unas cuantas, algunas con más éxito que otras, pero siempre con pundonor, sé que, de no ser por él, esto habría sido más difícil. Así que en estos momentos parte de mi gratitud va hacia él, espero que ambos podamos acabar esta guerra, aunque sea para empezar otras, lo más enteros posible. También quiero mencionar especialmente a David, que en esta última etapa ha sido con quien he compartido sensaciones y frustraciones, esfuerzos y dificultades, y por ello lo recordaré cuando recuerde este momento.

Además, por último, pero no por ello menos importante, y me permito esta frase tan manida por imprescindible, quiero agradecer la confianza de mis profesores, de todos ellos, aún recuerdo a los de hace dieciséis años, que se dice pronto. Amparo, Ricardo, Mari Ángeles, gracias por ayudarme cuando aún no nada conocía del mundo. Luis, Fermín, Honorina, Juan Carlos, Pedro, Amaya, Manuel, Piedad… en mayor o menor medida me habéis ayudado a crecer como persona, y me habéis acompañado en momentos importantes, fue en el instituto cuando decidí que me dedicaría a esto, lo que quería ser en la vida, y sin vuestra ayuda sé que habría sido mucho más difícil. No sabéis lo agradecido que estoy de haberme topado con vosotros, bendita suerte.

Ya más recientemente, no puedo dejar de agradecer a David Expósito haberme permitido formar parte de este proyecto, y haberme dado la confianza para realizarlo. Ha sido una experiencia enriquecedora y nunca lo olvidaré, pues ha sido uno de los momentos más trascendentales de mi vida, y estará marcado por su presencia. También quiero agradecer a José Daniel García la inspiración del mismo. En este momento recuerdo cuando los conocí a ambos hace unos años y, la verdad, uno no sabe cómo puede haber elegido estos derroteros para desarrollar su formación, dado que trabajos como este inspiran temor, no negaré que a mí me pasó, pero es también terriblemente satisfactorio dedicarse a tareas como esta.

Seguro que me dejo personas en el tintero, que me olvido imperdonablemente de alguien, pero ya he tenido que descansar varias veces para sosegar mis emociones y no creo que pueda seguir escribiendo más, sólo puedo decir que no cabe gratitud mayor que la que siento ahora mismo.

«La realidad es muy difícil de soportar para quienes creen que cualquier tiempo pasado fue mejor»

Juan Carlos I, Rey de España

Contenido

Índice de tablas

Índice de fragmentos de código

Índice de ilustraciones

Índice de requisitos

Índice de requisitos de usuario

Requisitos de sistema

Índice de casos de uso

Índice de pruebas

Índice de gráficos

Abstract

Introduction

Since the first supercomputers were built, the way to work with them was using an enormous set of microprocessors to do the work faster than using just one. This need has become imperative nowadays even in domestic computing because increasing the clock frequency in domestic processor has become impossible. This need for parallelism turned shared and distributed memory parallel programming models in a main problem in modern high performance computers.

The structure of this project, the motivation for this job and the basic concepts exposed in it will be described in this section. Also, a description of the motivation and the background that led to do this work is included.

Motivation

It is undoubtable: in actual high performance computing, a large amount of processor must be used, in the best situation, or even very many computers, connected by a network to do the work more efficiently. But the biggest problem when doing that is synchronizing and communicating the computers. It is a hard work to do, and not doing it will may lead the project to fail.

That's why computing using clusters is one of the standard ways to make a huge amount of processing in a system. And it is not used only when doing scientific work, it is also used when offering web services, managing a corporative network or even when processing online videogames data. On the other hand, we need to make this using the most efficient software we're able to achieve.

That is why two of the more used programming languages to write parallel programs are C and C++. The C family programming languages are known by their good performance. But these languages suffer a lack of distributed memory tools. And that is why they were made when computer clusters were not so needed. The principal motivation to do this work is, mainly, merge the powerful tools provided by C++ and the existent distributed memory frameworks to make possible programming simple computing in a distributed memory paradigm with the smallest user intervention. What does this mean? This means that the user of the software described in this project won't have to change its sequential code to adapt it to distributed memory.

Given that the problem scope is very large we've decided to focus our efforts in a little sequence of code, a concrete program type, that is usual in scientific computing. That program consists of: processing a vector with a set of algorithms. And it is so appropriate because one of the most important tools provided by C++ are containers. And the most used container class is the vector. It is used because it may grow efficiently if it is needed, it represents an ordered set of elements and, when used, it is a simple abstraction but powerful to manage a set of elements properly.

After this decision, we needed to choose a framework to make C++ distributed memory ready. In section 3 of this document we describe widely the different

alternatives and why we choose message passing as the best to do this work. Here, we're only saying the most important facts that contribute to do this the most proper election, concerning to us.

Message passing interface is a standard when doing distributed memory computing, and it has been a standard because it offers a simple way to communicate and synchronize processes. In fact, communicating and synchronizing processes are the main problems when developing distributed memory programs, and that is why distributed memory means that data is allocated in separated memory spaces, because we are using different processes, or even more than one computer. This makes processes unable to use simple memory access instructions to access data when it is not allocated in his own memory.

For performing remote memory access, the process who allocate the data needs to know, in some way, that it must send the data to the petitionary process, and the petitionary process needs to know where and how request the data.

Message passing interface offers a simple why to do that. When programming using this framework, the programmer writes one program, and in it he must discern which will be portion of data associated to each process in the program. Every process executing in a MPI application has a unique id. Using this id, the programmer can change the execution flux of every process using control structures. That way to work is the same used when using UNIX operating system calls to make a copy of the current process. This allows every process to know what it has to do. On the other hand, we still need a way to communicate processes, to exchanging the data.

The name of this framework is not casual, the way to work with message passing includes two basic operations: send a receive data. This ends the problem of communicating processes, because the process which sends the data and the process who receives it will execute send and receive functions depending on their ids. Additionally, we still need to solve the synchronization problem.

But send a receive functions solve this problem too, because they are synchronous. This means that when using one of them, the fact that the data has been received when the functions returns is guaranteed. The same happens when executing receiving functions. By communicating process, we are synchronizing them, making a process wait for the information calculated by other process in an implicit way to the receive operation.

But the most important thing that makes message passing a good tool to make distributed memory computing when it is used with C++ is that it has a data-centred way to work. Since other distributed memory solutions look at the operations as the main element of their way to work, message passing uses data as the base of the paradigm. And it is so convenient because we're trying to make a container work in a distributed memory context. Since containers are a way to store and order data and access it in an easy way, the main activity we will do will be copying data from one process to other and synchronize processes while they are transmitting that data between them.

In addition, if the sequential code we're trying to convert to distributed memory code is processing vectors, the data stored in these vectors must come from some place which is usually a file. We need to give the user a way to easily read a file into a

vector that will be distributed in more than one process or machine. And MPI, in one of its most used implementation, MPICH, offers a way to do this efficiently. MPICH offers tools to read or write data with concrete patterns, and this allows the programmer to create data structures (datatypes) to manage data in an efficient way and delegate the effort to translate these data structures to bytes in a file.

These are the reasons to make this work in this way. In chapter 3 we describe in more detail the reasons to choose this software.

Objectives

The main objective of the software described here is allow a programmer to use a code that looks like a sequential one in a cluster of machines, and take advantage of distributed computing in an easy way. Using a library to do this, we want to offer a distributed version of the standard C++ vector container and a set of algorithms to work with it, taking advantage of distributed memory parallelism. This main objective has a set of key parts:

- Allowing the user to instantiate a vector that will be stored in the memory of more than one process or machine.
    - And make it able to choose how will be the data distributed to processes per concrete patterns.
- Providing a way to read binary data from a file and get it into the vector, distributed as expected.
- Providing a set of algorithms, with the same interface that their STL versions, that make possible to compute with the data in the vectors.
- Providing a way to access and operate with the vector like C++ way to access a vector. It includes iterators, operators and other C++ conventions, widely explained in section 2.1.

This project has some secondary objectives too:

- Taking advantage of distributed computing to increase performance when using distributed algorithms.
- Implementing a load balance algorithm to make more efficient the execution of distributed algorithms.

Document structure

In this section a description of every chapter of this document will be shown:

1. Introduction: This section describes the project objectives and the motivation to do this work.
2. Short introduction to C++ and MPI: This section is a short explanation of key concepts of C++14 and MPI, to make easier for the reader to understand the paper if the reader is not familiar with these technologies.
3. State of art: This section describes different alternatives to implement this software and makes an analysis of C++ and its lack of distributed memory tools.
4. Development framework: This sections describes detailly which software has been used to develop this work and explains the hardware used to develop

and test the software. In addition, it makes an analysis of the legal constraints of this project.
5. Software description: This is the biggest section in the document, and it describes the software developed, including requisites, use cases and detailed design of the library.
6. Validation tests: This section enumerates the validation test done to this software and their results.
7. Performance tests: This sections describes performance test done to this software and includes an analysis of their results.
8. Socioeconomic impact: This section evaluates the economic impact of this software and makes a detailed budged description for this development.
9. Conclusions and future work: This sections makes a summary of which objectives of this project have been accomplished.
10. Bibliography

Summary

This project presents a library which permits users to make distributed computing without changing its code in a large extent. In this section, we will describe the design used to achieve that and a summary of tests done to probe if the software has accomplished the objectives.

First, we need to identify which elements of a C++ vector are the most important ones. After that, we need to implement that elements in a distributed memory context without changing user interactions with the classes. There are three key elements when using a vector in C++. A vector may be used with the access operator, which is the same that is used in C with a static vector. In addition, we need to implement an iterator for this distributed container which allows user to use it with STL algorithm and other standard C++ functions.

The first objective is knowing how to stores the data in the processes. The best way to store data in a C++ program is using a standard vector. And that is what we'll do. Every process will have a standard vector and will store its data there. Once the vector has been initialized, we know every information we need about the distribution mode.

After this, the user can allocate in the vector the elements from a raw binary file. To do that, it will provide the number of elements it wants to read. After this, we can read the elements from the file and distribute them between the processes, which will store them in local vectors.

Depending on what distribution is used, the elements are read in a concrete pattern, to do that, the program will create a derived MPI datatype (see **2.2**) to read them calling only once to the read function offered by MPI.

Once we have read the elements and they are allocated in processes, we'll need to implement a good way to access every element in the vector. The first way to access a vector is using the access operator.

To implement an access operator, we need to calculate the process which allocates the element that the user has required and it must send it to the other processes.

And we need to be able to allow the user use that function in the same way it uses the access operator when it's using standard C++ vectors.

If we want that access operator behaves the same way in every process, the process which is allocating the data must do a broadcast to the other processes. If we attend to MPI way to work, if every process executes the operator, we need to know which process stores the data. To do that, we can implement a function that, for a given position, its result is the process id that allocates that position. It will do that using the distribution information provided by the user when the vector's instantiated.

After that, we need to implement the iterators used with the vector, an iterator is an object which represents a pointer to an element in a container. In a vector, an iterator is a very simple data structure, because an element in a vector is represented by its position on it. Thus, an iterator must contain information about the vector it is pointing to and the position which it is referencing.

Once the iterator is designed, we need to implement every operation we can do with a standard vector C++ iterator, these operations are, mainly:

- Access the element by using the indirection operator over the iterator.
- Change the element which the iterador is referencing using increment operators.
- Provide a class function in the vector to get an iterator to the first element and other one to get the next element to the last.

To make the iterator able to access the element, we have a tool that may be used, we have an access operator which receives as argument the position we are accessing. The iterator contains a reference to the vector which it is pointing to. Using this reference, it is able to invoke the access operator using the number of the position it is pointing to, accessing in an easy way to the element.

In addition, increment an iterator is equivalent to add one to the position it is referencing. Finally, we can instantiate an iterator using a vector and a position, to get one that is pointing the first element in the vector, we just need to return an iterator with this vector and the position 0. The same thing is done to get an iterator to the element next to the last, but instantiating it with the size of the vector instead of zero.

After providing to users a way to work with this vector in the same way they do with a standard C++ vector, a set of distributed algorithms are also provided. To get this, we have inspired our functions in C++ STL functions, which are a generic way to express the algorithms.

The two algorithms chosen for this project are transform (which executes the same function over every element in a range of the vector) and reduce (which applicates an operator to all the elements in a range). The first function is the most difficult, that is why it may imply copying data from a vector to another, which is hard to do when using distributed memory, because these vectors may have different distribution models. For each element in range, we need to calculate which process allocates the data in the source vector and which process allocates the data in the destiny vector to allow them to make a point-to-point communication to copy that element.

To achieve this objective the function calculates which process allocates each part of the input data and which process allocates the output data. Then, both processes know what they must do, the process which allocates the data sends it to the process who need it to apply the function to the data and store it in its private vector.

Reduce function if less complex than transform. That function does not imply any data transmission between vectors. This function is going to be described with and example. A reduce operation applies an operator (which must be commutative and associative with itself), an example of this kind of operation is addition. If the vector we're applying reduce function to is: {1, 2, 3, 4, 5}, and we apply a reduce operation based on addition, the result would be $1 + 2 + 3 + 4 + 5 = 15$.

When the vector is distributed between a set of vector, every vector needs to calculate a partial reduction operating with the elements it has. After getting this partial result, MPI framework provides a collective operation to merge all partial results in an efficient way, applying the operator (in the example, addition) to the partial results. If a process hasn't got a partial result because it has not element to operate with, it won't participate in the merging operation.

One of the objectives of this project is to provide a set of procedures to allow users decide how the data will be distributed between processes. We have implemented three models to distribute data and a load balancing algorithm, which distribute the elements depending on the time every process lasted to execute the code the last time.

The first mode, and the simplest, is the block distribution. In this mode, each process receives a consecutive set of elements, whose length is equal. If the total element amount is not divisible between processes, the remainder, if it exists, is assigned to the last process, which is the only one which can allocate a different number of elements.

The second mode is a cyclic one. The size of the cycle is determined by the user, and the elements are distributed in this way: the vector is divided in portions, every portion is assigned to a process. The first portion is assigned to the first process, the second to the second and so on, when it arrives to the last process, it starts again from the first process. If the vector is not exactly divisible in portions, the last potion will be smaller than the others.

The third mode is a custom mode. In this mode, the user specifies the number of elements that every process will allocate. The vector will be distributed in blocks of these lengths, and every process will receive its corresponding portion.

Finally, for the load balance mode. After executing a performance test, registering execution time, when a user chooses this mode, the library will instantiate the vector in the custom mode, but the block lengths will be determined by the execution time in the test. Distributing the elements is performed inversely to the time wasted executing the program, in this way slower processes will receive less elements, and the fastest ones will receive more elements.

Once we've described all the operations we can do with the distributed vector and the distributions modes, other key point of this project is merging C++ characteristics with the C dependant interface of MPI. MPI uses C concepts to

manage its features, examples of this are pointers arrays, function pointers and not using exceptions to manage errors. To be able to use C++ concepts with the MPI interface, we need to convert some of this concepts to their C equivalent. For example, if we need to send a chunk of data using MPI, we need to convert it to a pointer, not to a class.

The most complicated example of this what converting every possible operation in C++ to a function pointer, which was needed to implement the reduce algorithm using the MPI reduce operation. To execute MPI reduce it is needed to create a MPI operation, which is created using a function pointer. To do this, it was needed to use metaprogramming techniques, including create an inner class of to the reduce function in the library. Using a template argument, we could create an inner class and get a pointer to a static function from it. Further description of this problem is in section **5.4.7**.

Now, we're going to describe the logical design of this software. In fact, it has a simply design because it has not a huge number of requisites. First, the software is implemented as a namespace, with allow us to present it like a homogenous component and adapts some names and variables to the library without changing their name. For example, the algorithms *transform* and *reduce* which can be named like the STL ones because they are in another namespace.

First, we need a set of data structures to store some important (and constant) data which we need to manage while the software is running. These data structures are:

- The performance data that will be used to execute load balance algorithm is stores in the same path, which is a string constant in our library.
- As we've said, in MPI every process has an identifier and knows the number of processes which are running this program, we'll store both data in a structure.
- A flag to control whether execution time must be stores or not, used when the user instantiates a vector with benchmark mode, allows to avoid erasing other benchmark data if the user does not want to replace it with this execution time.
- A stream variable, used to allow the user to print on screen without duplicating the output, making easier to it change the code from standard cout to our namespace cout.

There are six classes in our library, but the most important ones are *DistributedVector*, *iterator*, *ifstream* and the *initializer*.

Starting by the last of them, the initializer fulfils the function of giving the library the information it needs to start the MPI execution. When a MPI program starts, every process must execute MPI Init function, and it receives as argument argc and argv, the argument of main function. Due to that, the runtime needs to obtain this information. To achieve that, we need the user to instantiate an object of this class and pass to the constructor function argc and argv as arguments. In addition, that instance of initializer will register its instantiation time and its destruction time, the difference between these times will be the execution time, and will be registered in the file of performance information if required.

The ifstream class has just a member, which stores the path to the file which is opened in ifstream initialization. And it has two functions, read and write, which are used to read and write from the file pointed by the path member. These functions call to two DistributedVector member functions, which will be described later.

Distributed vector is the biggest and most important class in the library. It has four members: the vector who contents the data, the schedule data, a MPI datatype and a "dummy" member. The vector will content the data stored by this process according to the distribution mode. The schedule data is the data of the distribution mode (for instance, the size of the cycles if it is cyclic, etc.) The MPI datatype is used to have a quick reference to it when used to read the vector, because it will be used to write it, too. Finally, the "dummy" member is used in the access operator. It is used in order to being able to return a reference when executing access operator. When a process requires remote data, it receives that data from another process and stores it in this variable, then, returns a reference to the variable.

The functions in this class, apart from the public ones that implement operations (get begin and end iterators, access, instantiate it, etc.) are:

- Check type: it is an operation used to identify if this vector is of integers or double precision floating point number. It is used because some MPI functions needs to receive this result as an argument to receive, send and read and write data.
- Owner: this function says, for a given position, the process with allocates it and it is, thus, one of the most used functions in the library.
- Global to local position: when a process needs to access a position which it allocates, it receives the position in the global vector, which is distributed between the processes, but it needs to know where that element is in its local vectors, and that is what this function does. For a given position, gives the equivalent in the local vector of the process which stores it.
- The class has got three constructors in order to allow the user to instantiate it with the available distribution modes and provide the information needed in that mode (cycle size in cyclic distribution o block sizes in custom distribution, for example)
- The access operator which has been is already explained.
- Get and set, which are two operations that allow the user to get a data from the vector, but control which node the correct value is received in and set data with and operation only in the process which allocates the element.
- Size: it is a function that behaves the same that the size function in a standard vector, says how many elements the vector allocates.
- Fill and write, which are two functions that make the real task of read and write the vector from and to the disk. To read the vector from disk, the fill function takes the distribution mode information stores in the vector member. Then, it makes an MPI datatype that represents that pattern and uses MPI function to read it from disk. The write function uses the MPI datatype stored as a member and write the vector to disk using it.
- The vector has begin and end member function that behave the same that their standard versions, returning the iterator to the first element and the iterator to the element next to last one.

Finally, the iterator class implements the functions required by a forward iterator in C++. These operations are: reference, increment and comparison. Its members are only an integer, which represents a position in the vector and a reference to the vector which it is referencing. The related operations are:

- A constructor that allow to instantiate it with just the position and the referenced vector.
- A function to get the position, used by the algorithms reduce and transform to allow it access directly to the position without using the access operator, in order to avoid the big overload that it has.
- A function to get a reference to the referenced vector, used to the dame thing that the last one.
- In addition, it has all the operators needed to implement the main iterator characteristics.

Results, conclusions and future work

In this section, we're going to expose the technic concussion of this project and a summary of the performance test done to this software. In addition, we're going to explain what are the future work guidelines, according to us.

El main objective of this software was providing an interface that looks the same that the standard one used un C++ to manage vectors, but implementing in a distributed memory context. That task included allowing the user to use a version of the standard vector class to whose data was stores in more than one process or machine. That objective has been accomplished, because the interface o this library allows the user to choose the distribution mode and the data will be stores in the nodes which are running the software in a transparent way.

Other objective was to provide a mechanism to allow the user to read a file and allocate the data in a vector and write it back to another file. This objective was also accomplished. The standard way to fill a vector from a binary data file in C++ is using an object that allows to open a file, symbolizing the stream of bytes and reading these bytes to the memory allocated by the vector. This is no possible in a distributed memory context because the vector is not a memory chunk in just a process. Instead, an interface of another object was provided, including write and read functions and receiving the vector to write/read and the number of elements as arguments.

The next main objective was offering the user a set of distributed algorithms represented by functions, which allows to operate with the vector contents. These algorithms must have the same interface that the STL ones. The chosen ones to be included in this library was transform and reduce. The distributed version of this algorithms we have implemented allow the user to invoking them with any available invokable object in C++ language, including lambda expressions or function-objects which is a very important step to merge the MPI power with C++ versatility.

Finally, the last of the main objectives of the library is to permit the user to access the created vectors using the same syntax that is used in C++. This objective includes several tasks: use the access operator as it does in C++, use iterators and, in addition, make these elements compatible with C++ standard functions that use these elements.

The access operator behaves the same way the standard C++ one does, and it's capable of returning a reference. This is useful because it allows to make composed operation with the access operator and allows the user to pass as an argument the result of the access operator to functions that receives a reference as argument. The iterators are compatible with standard C++ functions such as advance (which increase the operator position n elements) or the STL functions. Thus, the objective of implementing an interface as close as possible to the standard C++ one is accomplished.

As a secondary objective, it was established that executing in a distributed memory context may achieve a better performance. In addition, a load balance algorithm which would allow the machines to use more efficiently their capabilities was added. To valuate if these objectives have been accomplished, a set of performance tests have been done, the summary of their results is:

- The performance of transform when it is used to apply a function to the whole vector and store the result in the same vector is good, which is a satisfactory result because it is a very common operation.
- Performance of reduce is not high, but we achieved user MPI Reduce with every invokable object in C++, with a big step in combining MPI and new C++ capabilities. In addition, it allows to perform this operation over vector that are not small enough to be stored in just a computer.
- Optimization that MPICH presents in read and write files allowed us to improve performance of the sequential version of the software.
- Thanks to MPI, transmitting data over the network does not produce a big overload if that network has a good bandwidth.
- The overhead of using access operator is high, so it is not possible to use it very many times without degrading the program performance. But it accomplishes its objective so well, because it is perfectly usable.

Future work

The interface provided by this library is as close as possible to the standard C++ one. The performance objective has been partially covered within this work, due to that, that is the main way that the future work must take. The first step would be to implement a better version of distributed algorithms. In the case of reduce, that work is easy, but in the case of transform, an algorithm which would allow to perform transform operations between vectors (those operations where first and last are not referencing the same vector that result) will be so complicated, and that is why it was not implemented at the beginning of the development. The main direction, according to us, that future work must follow is:

- Improve the performance of reduce and transform algorithms.
- Make an analysis of the present distributions modes and delete, improving some of them.
- Include more distributed algorithms to the library.

The actual implementation of the distributed algorithms is intuitive, because it uses the STL philosophy, making an evaluation of every element, but it is not efficient, that is why the future work must change that philosophy to new algorithms thought exclusively for performance. If the future works achieve this objective, this software

will become a very useful tool in distributed memory computing, offering a simple way to make C++ and its main characteristics take advantage of distributed memory parallelism.

1. Introducción

Desde que los primeros supercomputadores se construyeron, la manera de trabajar con ellos fue utilizar un gran conjunto de procesadores para hacer el trabajo más rápido que usando sólo uno. Esta necesidad se ha convertido en imperativa hoy en día incluso en entornos domésticos debido a que no se puede elevar más la frecuencia de los procesadores, haciendo necesario que incorporen más núcleos y que estos sean utilizados en cálculos paralelos. Esta necesidad de paralelismo tanto en memoria compartida como en memoria distribuida se ha convertido en uno de los problemas más importante en los ordenadores de altas prestaciones.

La estructura de este proyecto, la motivación de este trabajo y los conceptos básicos expuestos en él serán descritos en esta sección. Además, una descripción de los antecedentes que llevaron a este trabajo será incluida también.

1.1 Motivación

En la computación actual de altas prestaciones es necesario utilizar varios procesadores, en el mejor de los casos, o incluso varios ordenadores, conectador por una red para hacer el trabajo más rápidamente. Pero el mayor problema cuando se hace esto es sincronizar y comunicar los ordenadores. Es un gran trabajo que, si no se hace, llevará el proyecto al fracaso.

Por eso es que la computación usando clústeres de ordenadores se ha convertido en la forma estándar de hacer una gran cantidad de procesamiento en un sistema. Que no se utiliza sólo en computación científica sino también en servicios web, manejando una web corporativa o procesando los datos de videojuegos en red. Por otro lado, necesitamos hacer esto con el *software* más eficiente que podamos conseguir. Utilizar una gran cantidad de ordenadores con un coste excepcional para malgastar sus capacidades con un pesado *middleware* no tendría sentido.

Por este motivo dos de los lenguajes de programación más usados son C y C++. La familia de lenguajes de C es conocida por su buen rendimiento. Pero estos lenguajes sufren de una falta de herramientas para la memoria distribuida. Esta situación viene dada, en parte, porque fueron creados cuando las necesidades del mercado tecnológico eran otras. La principal motivación para realizar este trabajo es, aunar las herramientas ofrecidas por C++ y los *frameworks* existentes de memoria distribuida. El objetivo final sería que el programador no tuviera que cambiar su código secuencial para trabajar en memoria distribuida.

Pero ese objetivo sería muy grande para un proyecto como este. Debido a esto, se ha decidido centrar los esfuerzos en una pequeña secuencia de código, un tipo concreto de programa, muy común en computación científica. Ese programa es: procesar un vector con un conjunto de algoritmos. Y es apropiado porque una de las herramientas más importantes de C++ son los contenedores. Y el contenedor más usado es el vector. Es usado porque puede crecer de una manera eficiente si es necesario, representa un conjunto ordenado de elementos y cuando se utiliza, es una abstracción simple pero capaz de manejar un conjunto de elementos apropiadamente.

Después de la decisión, es necesario elegir qué *framework* dotaría mejor a C++ de capacidades de memoria distribuida. En la sección 3 de este documento se describen las diferentes alternativas y por qué se ha elegido el paso de mensajes como la mejor para este trabajo. Aquí, sólo se mencionarán las características más importantes del mismo que han contribuido a la elección.

La interfaz de paso de mensajes (MPI) es un estándar cuando se realizan tareas de computación distribuida, y ha sido así porque ofrece una manera simple de comunicar y sincronizar procesos. De hecho, cuando se trabaja en un paradigma de memoria distribuida, el problema principal es la sincronización y la comunicación de los procesos. Esto es así porque no se pueden utilizar meras operaciones de acceso a memoria para acceder a los datos. Dichos datos están almacenados en espacios de memoria diferentes y, en muchos casos, en ordenadores diferentes.

Esto provoca que dos procesos pueden intervenir en una operación de acceso a memoria, el proceso que tiene el dato debe saber, de algún modo, que debe enviar el dato al proceso que lo requiere y, el proceso que lo requiere tiene que saber qué proceso se lo debe enviar.

MPI ofrece una manera sencilla de hacer esto. Cuando se programa utilizando esta plataforma, el programador debe escribir un solo código y en él debe discernir qué parte será ejecutada por cada proceso. Cada proceso ejecutando en una aplicación MPI tiene un identificador único. Usando este identificador, el programador puede cambiar la ejecución de cada proceso utilizando estructuras de control. Esa manera de trabajar es la misma que la utilizada en un sistema UNIX cuando se crea una copia del proceso actual para ejecutar tareas en paralelo. Por otro lado, necesitamos una manera de comunicar los procesos, ahora que ya saben qué deben hacer.

El nombre de MPI no es casual, la manera de trabajar con el paso de mensajes incluye dos operaciones básicas, enviar y recibir datos. Esto termina el problema de la comunicación, porque los procesos que deban enviar o recibir datos lo sabrán basándose en sus identificadores. Puede mandarse cualquier tipo de dato. Pero aún queda resolver el problema de la sincronización.

Pero enviar y recibir mensajes soluciona este problema también. Estas operaciones son síncronas, de tal modo que cuando se usa una de ellas, no se retorna de la misma hasta que los datos han sido recibidos. Comunicando los procesos, también se sincronizan, haciendo que la espera de un proceso por la información que ha pedido sea implícita a la recepción de la información.

Pero el hecho más importante que convierte el paso de mensajes en la mejor herramienta para dotar a C++ de capacidades de computación distribuida es que es un modelo centrado en los datos. Otros modelos de computación distribuida sólo se centran en las operaciones. Pero el paso de mensajes se centra en los datos, lo cual es una ventaja cuando se quiere implementar un contenedor en memoria distribuida, debido a que un contenedor no es más que una abstracción para organizar y acceder a información de un modo estructurado.

Además, si el código secuencial que queremos convertir a uno que se ejecute en un contexto de memoria distribuida es el procesamiento de vectores, los datos que llenen esos vectores deben venir de algún sitio. La manera más sencilla de acceder a una gran cantidad de datos es leer un fichero. Necesitamos dar un mecanismo al usuario

para leer un fichero de un modo sencillo y distribuir esos datos entre los procesos. Y MPI, en una de sus implementaciones más utilizadas, MPICH, ofrece herramientas para hacer esto de un modo eficiente, dado que dispone de un concepto llamado tipos de dato (*datatypes*) que permite la lectura de un fichero con determinado patrón simbolizado en una estructura de datos.

Estas son las razones para hacer este trabajo de este modo. En la sección 3 se detallan más las razones para escoger este software para la realización del trabajo.

1.2 Objetivos

El objetivo principal de este software es permitir a un programador utilizar un código que parezca el que se usaría en un programa secuencial para ejecutarlo en un clúster de máquinas. Por otro lado, utilizando esta librería, queremos ofrecer una manera al usuario de utilizar una versión distribuida del contenedor vector y un conjunto de algoritmos que trabajen con él, sacando provecho de las ventajas de la computación distribuida y del paralelismo. El objetivo principal tiene una serie de elementos clave:

- Permitir al usuario instanciar un vector que será almacenado en la memoria de más de un proceso u ordenador.
    - Hacerlo capaz de elegir como esos datos serán distribuidos a los procesos con patrones concretos.
- Ofrecer una manera de leer datos binarios desde un archivo y usarlos para llenar el vector, distribuidos como se especificó.
- Ofrecer una serie de algoritmos, con la misma interfaz que sus versiones de la STL que hagan posible hacer cálculos con los datos del vector.
- Ofrecer una manera de acceder y operar con el vector como se hace con un vector estándar de C++, lo cual incluye iteradores, operadores y otras convenciones de C++, ampliamente explicadas en la sección **2.1**.

Este proyecto también tiene estos objetivos secundarios:

- Sacar provecho de la computación distribuida para aumentar el rendimiento cuando se utilicen algoritmos distribuidos.
- Implementar un mecanismo de balanceo de carga que haga más eficiente la ejecución de los algoritmos en un conjunto de máquinas con diferentes características.

1.3 Estructura del documento

En esta sección se describirá cada capítulo del documento:

1. Introducción: Esta sección describe el proyecto, sus objetivos y la motivación para hacer este trabajo.
2. Breve introducción a C++ y MPI: Esta sección es una pequeña explicación de conceptos clave de C++14 y MPI. Para hacer más fácil para el lector entender el documento si no le son familiares estas tecnologías.
3. Estado de la cuestión: Esta sección describe las diferentes alternativas para implementar este software y hace un análisis de C++ y sus carencias de características de herramientas para la ejecución en memoria distribuida.

4. Entorno de desarrollo: Esta sección describe detalladamente qué *software* ha sido utilizado para desarrollar este trabajo y explica el *hardware* utilizado para desarrollar y probar este *software*. Además, hace un análisis de las restricciones legales del *software* y del marco regulador del mismo.
5. Descripción de la biblioteca: Es la mayor sección del documento y describe el *software* desarrollado, incluyendo requisitos, casos de uso y el diseño detallado de la biblioteca.
6. Pruebas de validación: Esta sección enumera las pruebas realizadas a este *software* para comprobar que cumple sus requisitos y los resultados de dichas pruebas.
7. Evaluación de rendimiento: Esta sección describe las pruebas de rendimiento hechas al software e incluye un análisis de sus resultados.
8. Impacto socioeconómico: Esta sección evalúa el impacto económico de este software y hace una descripción detallada del presupuesto para su desarrollo.
9. Conclusiones y trabajo futuro: Esta sección hace un resumen de qué objetivos de este proyecto han sido cumplidos y de qué objetivos deberían tener trabajos futuros que tomen este como base.
10. Bibliografía

2. Breve introducción a C++ y MPI

Dado que este TFG propone diseño a bajo nivel en C++ y MPI hemos considerado oportuno hacer una breve introducción a los conceptos clave del lenguaje que serán utilizados en este proyecto y a la biblioteca. Si ambos conceptos son familiares para el lector, puede saltar este capítulo.

2.1 Introducción a C++

C++ es uno de los lenguajes más populares en la actualidad [25]. Esto es debido, además de a su relación con C y con el desarrollo de software de sistemas, a la versatilidad del mismo. C++ cuenta con herramientas para programar en diversos paradigmas y muchas de dichas herramientas tienen presencia en este desarrollo. Los principales conceptos propios del mismo que aparecerán en este trabajo serán: la metaprogramación basada en plantillas, STL, el concepto de objeto invocable, sobrecarga de operadores, los espacios de nombres (*namespaces*) y el concepto de referencia.

Una de las herramientas más importantes con las que cuenta C++ es la metaprogramación. El concepto, de manera concisa es: «Escribir programas que generen otros programas» [26], es decir, generar un código que, a su vez, sea capaz de generar otro que cumpla otra finalidad. En C++, el principal elemento de metaprogramación son las plantillas. Una plantilla es, como su nombre indica, una serie de instrucciones para generar, a su vez, dicho código. Las plantillas principales son las plantillas de clase y las plantillas de función. Una plantilla de clase se escribiría como una clase en cualquier lenguaje orientado a objetos, pero, antes de la misma, se declararían una serie de símbolos que serían los que nos permitirían la creación de varias versiones de esa clase a partir de dicha plantilla.

El ejemplo más prototípico de esto y que, además, utilizaremos, en un contenedor. Un contenedor es una clase destinada a crear una estructura de datos que organice, a su vez, otras estructuras. Esto provoca que, si queremos que la solución sea versátil, deba poder hacerse una versión de dicho contenedor para cada estructura de datos que éste pudiera almacenar, imaginemos una lista enlazada definida de la manera más simple.

```
class nodo{
      int elemento;
      nodo next;
}

class lista{
      nodo primero;
      int size;
}
```

Código 1: Ejemplo de lista enlazada sin plantillas

Esta lista es la versión para almacenar datos enteros, pero habría que programar listas iguales para otros tipos de datos y, además, no se podría dar soporte a cualquier tipo de dato que definiera el usuario. Una solución si se hiciera en lenguaje C sería utilizar direcciones de memoria y el tamaño del tipo de dato. Los punteros son una herramienta potente, pero con un nivel de abstracción bajo que puede

inducir a errores. Definiendo la clase *nodo* como una plantilla de clase, sin embargo, podemos evitar este problema:

```
template <class T>
class nodo{
	T elemento;
	nodo<T> next;
}
template <class T>
class lista{
	nodo<T> primero;
	int size;
}
```

Código 2: Ejemplo de metaprogramación con plantillas

Al definir una plantilla de clase, establecemos una estructura que será completada por el compilador (y, por tanto, en tiempo de compilación). Lo que permite que, para cada tipo de dato que necesitemos, podamos generar una versión de la lista. Esto se logra mediante el argumento de la plantilla (*template argument*), que en este caso sería el símbolo *T*. En la definición de la plantilla *nodo* se define que se recibirá como argumento de plantilla el nombre de una clase. En la definición de la plantilla *lista*, a su vez, recibimos otro argumento de plantilla, y con él, instanciamos un nodo. Ese argumento, indicado en la declaración del miembro *primero*, provocará que el compilador genere la versión de la clase *nodo* que necesitemos para el tipo indicado como argumento de plantilla.

Además, las plantillas también son aplicables a las funciones, generando plantillas de función. Por ejemplo, el algoritmo de ordenación de burbuja (*bubble sort*) podría implementarse como:

```
void bubbleSort(vector<int> lista){
	for(int jj = 0; jj < lista.size(); jj++){
		for(int ii = 0; ii < lista.size()-1; ++ii){
			if(lista[ii] > lista[ii+1]){
				int aux = lista[ii+1];
				lista[ii+1] = lista[ii];
				lista[ii] = aux;
			}
		}
	}
}
```

Código 3: Ejemplo bubble sort *sin metaprogramación*

Estando en el mismo caso que en el de la lista, esta operación tiene sentido para cualquier tipo de dato para el que tenga sentido la operación de comparación, y, es más, tiene sentido para cualquier operación binaria cuyo resultado sea un valor lógico (cierto o falso). La metaprogramación permite generalizar el algoritmo:

```
template <class T, class Operador>
void bubbleSort(std::vector<T> lista, Operador op){
	for(int jj = 0; jj < lista.size(); jj++){
		for(int ii = 0; ii < lista.size()-1; ++ii){
			if(op(lista[ii], lista[ii+1])){
				int aux = lista[ii+1];
				lista[ii+1] = lista[ii];
				lista[ii] = aux;
			}
		}
	}
}
```

Código 4: Ejemplo de bubble sort *con metaprogramación*

De este modo, cualquier lista de elementos podrá ordenarse de acuerdo a un criterio proporcionado por el usuario (orden ascendente, orden descendente, comparación lexicográfica, etc.)

Es precisamente ese ánimo para generalizar algoritmos a cualquier contenedor y a cualquier tipo de dato lo que motiva la creación de la biblioteca estándar de plantillas (*standard template library*, STL). Es decir, el objetivo de encontrar la representación más general de los algoritmos [27]. Y la metaprogramación es una herramienta adecuada para lograr que los algoritmos sean representados de una manera genérica, como en el ejemplo mostrado en el Código 4.

Una manera de generalizar los algoritmos es, también, hacerlos independientes del contenedor. En el ejemplo anterior se especifica que ha de ser un vector, pero una lista enlazada es lo mismo que un vector, sólo cambian detalles de la implementación, pero no el concepto que representan: un conjunto de elementos ordenados. Para evitar utilizar funciones específicas para cada contenedor existe el concepto de iterador.

Un iterador es un objeto que sirve para referenciar a un elemento de un contenedor y, si fuera necesario, puede ser cambiado para referenciar a otros elementos. El ejemplo más simple sería un puntero en lenguaje C. Si tenemos un vector v, se puede usar un puntero para referenciar a los elementos del mismo y, mediante la indirección, se puede acceder al elemento. Si se desea, puede usarse aritmética de punteros para cambiar la posición referenciada. Un iterador no es más que la extensión de este concepto a contenedores diferentes y con más características, sobre todo se seguridad. Siguiendo con el ejemplo anterior, si usáramos iteradores:

```
template <class iterador, class Operador>
void bubbleSort(iterador start, iterador end, Operador op){
	for(auto ii = start; ii != end; ++ii){
		for(auto jjMax = (end-1), jj = start; jj != jjMax; ++jj){
			if(op(*jj, *(jj+1))){
				auto aux = *jj;
				*jj = *(jj+1);
				*(jj+1) = aux;
			}
		}
	}
}
```

Código 5: Bubble sort *implementado con iteradores*

De este modo, usando una sintaxis que sería familiar para un programador en C, se puede implementar el mismo algoritmo para cualquier contenedor. Este tipo de implementaciones son típicas en la STL, en la que los argumentos recibidos por la mayoría de algoritmos son iteradores, y no contenedores directamente. Éste es el caso de funciones como: *transform*, *reduce*, etc.

El operador de indirección para un iterador es, por convenio, el asterisco, al igual que para los punteros. Esto es posible porque C++ permite definir operadores para las clases como si fueran funciones miembro. Esto permite, por ejemplo, implementar una clase *fecha* que pueda ser incrementada con el operador ++ y que siga el comportamiento deseado por el programador.

En los ejemplos anteriores, el uso de metaprogramación ha permitido que una función reciba como último argumento otra función. Realmente *op*, podría ser cualquier tipo o clase para el que la sentencia «op(a, b)» tuviera sentido. En C++, los objetos que pueden ser invocados con el operador paréntesis son, fundamentalmente estos 3:

- Punteros a función: son el mismo concepto que en C, el nombre de una función es su dirección de memoria y se puede invocar con los argumentos que se definieron con la función.
- Un objeto con operador paréntesis sobrecargado: Todas las clases pueden implementar un prototipo o varios de este operador, con los argumentos que el programador elija.
- Una función lambda: Estas funciones anónimas son definidas en la ubicación donde se invoca o se pasa como argumento a otra función. Se usan para encapsular métodos que se pasan a algoritmos o a procedimientos. [28]

Sabiendo esto, existirían tres maneras de invocar a la función que hemos desarrollado:

```
template<class T>
class sorter{
public:
      bool operator()(T a, T b){
            return a < b;
      }
};
template<class T>
bool lessThan (T a, T b){
      return a < b;
}

int main(int argc, char const *argv[])
{
      std::vector<int> v{50,12,27,32,25,54,98};
      //puntero a función
      bubbleSort(v.begin(), v.end(), lessThan<int> );
      //objeto con operador sobrecargado
      bubbleSort(v.begin(), v.end(), sorter<int>{/*constructor*/});
      //función lambda
      bubbleSort(v.begin(), v.end(), [](int a, int b){return a<b;});
      return 0;
}
```

Código 6: Ejemplo de uso de objetos invocables

Por otro lado, un *namespace* es una manera de agrupar elementos en un paquete, bajo un nombre, para poder organizarlos. Por ejemplo, si un sistema quisiera implementar su propia versión de la clase vector, en un lenguaje como C tendría que cambiar el nombre a <nombre de la biblioteca>_vector o algo similar. Sin embargo, en C++ podría crear un *namespace* para solucionar la ambigüedad. Éste es el caso del espacio de nombres estándar (std), que, a su vez, contiene una serie de variables y clases para uso del programador.

Finalmente, C++ dispone del concepto de referencia. Una referencia es un símbolo que se comporta como aquél del que tiene valor. Permite no utilizar punteros para pasar argumentos a funciones con el fin de que éstas modifiquen el contenido. Además, permite que las funciones devuelvan esas referencias, haciendo que el resultado de una función se comporte como la variable cuya referencia ha devuelto. Por ejemplo:

```
int main(int argc, char const *argv[]){
	std::vector<int> v{1,2,3,4,5,6,7,8,9};
	std::vector<int>& vref = v; //& indica referencia
	v.push_back(10); //tanto v como vref tendrán un elemento más
	return 0;
}
```

Código 7: Uso simple de referencias en C++

En el caso de que una función tuviera que modificar uno de sus argumentos:

```
//Introduce en v los n primeros números naturales empezando en 1
//v es recibido como referencia
void naturalNumbers(std::vector<int>& v, int n){
	for(auto ii = 1; ii <= n; ++ii){
		v.push_back(ii);
	}
}

/*...*/
std::vector<int> v{};
naturalNumbers(v, 10); //la llamada a la función es igual
```

Código 8: Uso de referencias como argumento

Finalmente, una referencia que sea devuelta permite que ese valor sea usado exactamente igual que la variable a la que referencia:

```
//Introduce en v los n primeros números naturales empezando en 1
//v es recibido como referencia
//ahora v es devuelto como referencia.
std::vector<int>& naturalNumbers(std::vector<int>& v, int n){
	for(auto ii = 1; ii <= n; ++ii){
		v.push_back(ii);
	}
	return v;
}

int main(int argc, char const *argv[]){
	std::vector<int> v{};
	//el resultado de la función es el propio vector modificado
	auto it = naturalNumbers(v, 10).begin();
	for(;it != v.end(); ++it){
		std::cout << *it << std::endl; //imprimir los elementos
	}
	return 0;
}
```

Código 9: Uso de referencias como retorno

Para una lectura más detallada del lenguaje se pueden consultar [1] y [33]. Además de las referencias del mismo en *cplusplus.com* y *cppreference.com*.

2.2 Introducción a MPI

MPI (*message passing interface*) es una biblioteca estandarizada de código abierto (disponible en varias implementaciones) que permite la programación de aplicaciones que hacen uso de memoria distribuida. Utiliza el paso de mensajes como enfoque para dicha tarea. Un programa en MPI se ejecuta en uno o varios procesos, que pueden estar, o no, en varias máquinas. Su enfoque es que todos los procesos ejecuten el mismo código y, basándose en un identificador único asignado a cada uno, cambien su comportamiento para colaborar de manera efectiva.

Ese identificador suele ser denominado rango (*rank*), y las primitivas más básicas que ofrece esta herramienta son el envío y la recepción de mensajes. Estos mensajes son los que permiten la comunicación entre procesos. El envío y la recepción son bloqueantes, lo que les otorga la capacidad de ser usados como primitivas de sincronización entre procesos.

Además, MPI dispone de una colección de funciones para el manejo concurrente de sistemas de ficheros, permitiendo no sólo semánticas para el acceso concurrente a archivos sino también la elaboración de tipos de datos derivados que sirven para crear abstracciones más complejas que las del fichero como *stream* de *Bytes*. Estos tipos de datos nos permiten la creación estructuras complejas que podemos mapear en el archivo y que el proceso utilizará para acceder a él.

Por ejemplo: si queremos que un proceso sólo lea un número de *Bytes* iniciales de un archivo, podemos crear un *datatype* que simbolice ese bloque de datos. Así, podemos hacer uso de las operaciones de lectura sin tener en cuenta más que cuántos datos queremos leer.

MPI ofrece una colección de *datatypes* básicos que se mapean, en su mayoría, con tipos de dato del lenguaje C, ejemplo de ello son: MPI_CHAR, MPI_SHORT, MPI_DOUBLE o MPI_INT. Estos tipos de datos, a su vez, pueden ser agregados en

otros más complejos, utilizándose así para poder leer un fichero con determinado patrón (por ejemplo, sólo los *Bytes* en posición par). Esto permite realizar operaciones de lectura invocando una sola vez a la función de lectura, evitando por tanto sobrecarga en el sistema.

Un ejemplo sería la creación de un tipo de dato que simbolizara la lectura de los 10 primeros enteros en posición par de un archivo. Si empezamos a contar las posiciones en 0:

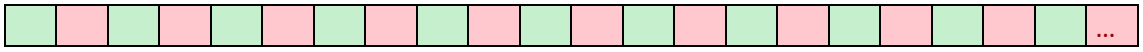


Tabla 1: Ejemplo de patrón de lectura de archivo

En el Ejemplo de patrón de lectura de archivo podemos ver en verde los enteros que queremos que el proceso lea. Uno de los *datatypes* derivados disponibles en MPI es el tipo de bloque indizado (*indexed block datatype*). Nos permite crear un tipo de dato como agregación de bloques de igual tamaño de un tipo de dato anterior (ya sea simple o derivado). Este tipo de dato se crea especificando:

- Cuántos bloques queremos
- Su longitud
- El desplazamiento de los mismos desde el inicio del archivo
- El tipo de dato que queremos agregar, en este caso MPI_INT.

Utilizando como parámetros: 10 bloques de longitud 1 que se desplacen los elementos requeridos (0, 2, 4, 6 …) podremos leer el patrón escogido sin necesidad de ocuparnos del tamaño del tipo de dato simple (del entero) ni de utilizar la función de lectura varias veces para poder leer los archivos con estos patrones. Así, invocando a la función de lectura para leer 10 datos de tipo MPI_INT, podremos leerlos con el patrón especificado.

Este tipo de operaciones están optimizadas en la biblioteca, permitiendo el uso eficiente de recursos del sistema y no realizando más llamadas al sistema que las necesarias **[11]**.

3. Estado de la cuestión

A la hora de hablar del desarrollo de herramientas para la programación distribuida en C++ y de las limitaciones o ventajas que éste pudiera tener para implementar este tipo de aplicaciones, primero hay que poner en contexto para qué fue diseñado el lenguaje y en qué momento; además, qué otros lenguajes usados ampliamente tienen soluciones más integrales para la programación distribuida. Hay que hacer por tanto énfasis en el contexto tecnológico y temporal del lenguaje y de sus influencias y requisitos.

C++ surge en los laboratorios Bell de manos de Bjarne Stroustrup (Dinamarca, 1950). En sus propias palabras, diseña el lenguaje como una extensión de C que permitiera que «Mis colegas y yo no tuviéramos que programar en ensamblador, C, o varios lenguajes modernos de alto nivel»**[1]**. El propósito principal del lenguaje era hacer para el programador individual más fácil producir buenos programas. Esto nos da una idea de que el origen del origen de C++ (parecido al del propio C) fue un lenguaje generado por una sola persona (o un reducido conjunto de ellas) y que tenía como objetivo que un programador pudiera generar código eficiente.

Esto tuvo una serie de implicaciones en desarrollo del propio lenguaje y en la filosofía del mismo. C++ fue concebido, y aún hoy se considera así, como un superconjunto de C. Esto hace que en principio C++ sea un lenguaje con características de bajo nivel, pero con mayor abstracción y con soporte para programación estructurada. Pero C++ nació con la firme vocación de añadir, al menos, el paradigma de orientación a objetos a la filosofía de C, lo que le hizo ser influido por Algol68 y Simula67 **[1]**. Estos lenguajes son orientados a objetos y éste es el paradigma que más se adapta a C++, pese a que soporte bien otros como la programación genérica o la funcional.

Debido a estas influencias, el lenguaje no se desarrolló en la tradición de otros lenguajes creados en la década de los ochenta o posteriormente, en que la programación concurrente era más sencilla, o en los cuales se daba soporte a programación distribuida, como por ejemplo Erlang **[2]**. En el siglo XXI se han creado lenguajes con abstracciones dedicadas a la computación distribuida tales como Go **[3]**. Pero C++, como hemos visto, no tenía como objetivo la programación concurrente. Éste es uno de los motivos por los cuales no existe un soporte nativo en C++ para la programación distribuida.

Además, por poner un ejemplo de un área relacionada, hasta la publicación del estándar C++11 no se creó una interfaz global para el manejo de hilos, es decir, de programación concurrente en memoria compartida, por ello, la solución era dependiente de la plataforma hasta que se creó la interfaz *std::thread*. Esto es explicado por el momento de desarrollo del sistema y sus aplicaciones iniciales, que son similares a las de C, si se desarrolla un *software* para un sistema (ya sea un sistema operativo o un sistema empotrado) precisamente el manejo de hilos será propio del sistema y deberá ser implementado. Con la extensión del uso del lenguaje para tareas de más alto nivel, el manejo de hilos de hizo independiente de la plataforma.

El lenguaje, además, fue creado en los inicios de la década de los 80, cuando la tecnología multiprocesador sólo estaba concentrada en unos cuántos centros de investigación y la capacidad de realizar cálculos en máquinas conectadas por red era

menor, puesto que las redes de conexión eran de menores prestaciones. Esta situación tecnológica hizo que no fuera especialmente contemplado el fenómeno de la programación concurrente, ni en memoria compartida ni en memoria distribuida. Lo cual es una de las mayores críticas al lenguaje respecto a otros que sí contaron con soporte nativo para programación en red, éste es el caso de Java, que tuvo soporte para ello desde el inicio de los noventa **[4]**.

A día de la fecha la programación concurrente con memoria compartida en C++ está más que superada, siendo una de las soluciones más adoptadas por su facilidad de uso y eficiencia OpenMP (*Open Multi-Processing*), debido a que es portable, está basado en directivas y es multilenguaje. Su éxito en C++ viene, además, de esa falta de estandarización de la programación multi-hilo hasta el estándar de 2011 (y, además, hay que considerar el lapso de tiempo desde que el estándar es liberado hasta que es implementado totalmente en compiladores). De hecho, si el estándar C++14 fue publicado el 2 de marzo de 2014 **[16]**, el compilador GCC no utilizó por defecto este estándar hasta la versión 6.1, liberada en 2016.**[17]**

Si bien las soluciones anteriores para la programación en memoria compartida (*multihreading*) son muy solventes, es muy interesante adaptar las aplicaciones a un paradigma de memoria distribuida. El interés de esto viene dado porque permite aumentar la potencia de un sistema añadiendo equipos de coste menor que un gran equipo muy potente. Esto permite aprovechar las capacidades de expansión horizontal del sistema. **[15]** La memoria distribuida, pese a tener un mayor coste de desarrollo, por su complejidad, ofrece la capacidad de conectar varias unidades de *hardware*, incrementando fácilmente las capacidades del sistema. Esto permite, por tanto, un aumento de la escalabilidad.

Sin embargo, pese a sus ventajas, la computación distribuida no tiene una solución estandarizada o integrada totalmente con el lenguaje. Hay muchas soluciones para implementarlo, pero todas ellas son bibliotecas o *frameworks* con sus propias peculiaridades, las principales alternativas para implementar un programa en un sistema con memoria distribuida en C++ serían: *sockets*, servicios web, RPC y MPI.

En un entorno de computación distribuida, siempre se debe tener en cuenta que esto implica que varios procesos ejecutarán sin compartir memoria. Si bien esto puede darse en un mismo computador, si la solución siempre se ejecutara en ese entorno, se podrían utilizar mecanismos de compartición de memoria entre esos procesos. El caso más prototípico es aquél en que los procesos ejecutan en máquinas separadas conectadas por una red de comunicación.

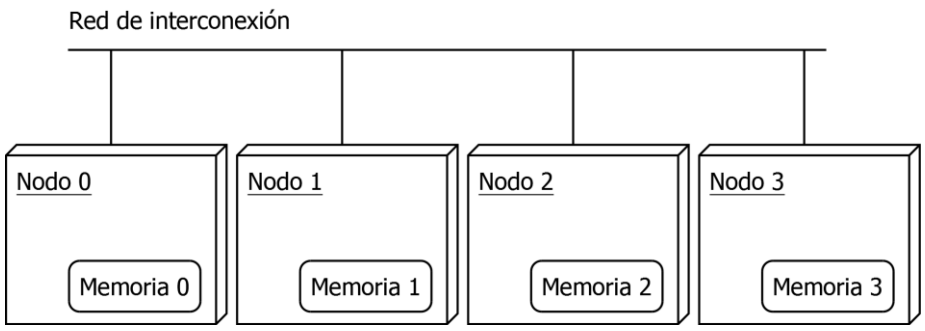


Ilustración 1: Esquema de memoria distribuida

Como se puede ver en la Ilustración 1, los nodos del clúster no pueden comunicarse entre ellos mediante el envío de direcciones de memoria o mediante operaciones sobre la misma, la información que compartan debe ser explícitamente comunicada a través de la interfaz de salida (generalmente una tarjeta de red IP) y el otro nodo debe recibirlo y poder interpretarlo. Esto tiene varias implicaciones:

- Los datos deben ser serializados para hacerse independientes de la plataforma. Deben convertirse de *Bytes* en memoria a un formato transmisible por la red y entendible en todos los nodos.
- No se pueden utilizar métodos de sincronización basados en memoria compartida (cerrojos o semáforos).
- Se debe utilizar un protocolo de comunicación.
- Los nodos deben disponer de un servicio de nombres.

Todas las soluciones que existen hasta el momento solucionan de una manera u otra estos inconvenientes. La primera de ellas, los *sockets*, lo hacen de este modo: la interfaz de un *socket*, en su versión más simple es como un archivo de texto plano, en el cuál se pueden escribir *Bytes* y que serán recibidos en el otro lado si se mantiene abierta la conexión. Todo esto asumiendo que utilizamos *sockets* IP de tipo *stream*. Son los más apropiados porque aportan fiabilidad en la comunicación y crear un canal de comunicación *Byte* a *Byte* entre dos nodos.

Aparte de crear el *socket*, conectarnos y escribir y leer datos, no nos ofrecen ningún servicio añadido, carecemos de serialización de datos, de servicio de nombres y registro o una abstracción de más alto nivel que las direcciones IP. Por ello, esta solución implica un desarrollo *ad-hoc* para cada aplicación, en la que compartir un dato entre dos nodos implica la apertura de una conexión (al menos la primera vez), el envío de los datos y, finalmente, su tratamiento. Esto presenta los siguientes inconvenientes.

El primero es que no es transparente para el usuario, puesto que tiene que enviar y recibir los datos a mano, ocupándose de que los nodos interpreten los mismos de la manera adecuada, lo que rompe con el concepto de comunicación transparente que sería deseable cuando se programa en sistemas de memoria distribuida. La implementación de los mismos es dependiente de la plataforma, es decir, hace que el código, en principio, no sea portable entre sistemas operativos de familias diferentes, dado que el servicio de *sockets* depende del sistema y de su interfaz **[5]**. No se debe puede saber qué nodos hay en el sistema de una manera sencilla, es trabajo del administrador aportar un mecanismo de descubrimiento, mensajes especiales dentro del protocolo o archivos de configuración para conocer sus direcciones.

Todos estos motivos dejan a los *sockets* como una alternativa muy primitiva para implementar operaciones en un sistema de memoria distribuida que sólo debería usarse si las demás no se adaptaran a los requisitos del mismo.

La siguiente alternativa es el uso de servicios web (*webservices*) para la tarea. Esta tecnología utiliza el protocolo HTTP para el transporte y el formato XML (o más recientemente, JSON) para la serialización de los datos. Esto ya aporta dos ventajas sobre los *sockets*, nos aporta una serialización automática de los datos y un servicio de nombres, puesto que al utilizar el protocolo HTTP podemos utilizar un servicio de nombres basado en URI (*uniform resource indentifier*). Éstos proporcionan un

esquema de nombres que puede nombrar la ubicación de un recurso o el nombre del mismo (URL o URN) [6] y que permite resolver, por tanto, la ubicación de operaciones o datos en nuestro sistema. Dependiendo de si los *webservices* se implementan de una manera más tradicional o como microservicios, se pueden utilizar codificaciones basadas en SOAP (que usa XML) o REST (que usa JSON), en ambos casos, se codifican todos los datos binarios a texto, de tal modo que un entero, a la hora de ser transmitido, en vez de ocupar, por ejemplo, 4 *Bytes,* ocuparía tantos *Bytes* como caracteres ocupara escrito. Además, se utilizan caracteres de control que añaden sobrecarga. Es muy versátil (muchos de los *frameworks* aportan serialización automatizada) pero añade mucha sobrecarga en la codificación y control a la red.

Los servicios web, se basan en el protocolo HTTP, este protocolo es sin estado y no orientado a conexión, lo que quiere decir que cada vez que un nodo quiera comunicarse con otro, mandar un mensaje, deberá abrir una conexión que se cerrará después de terminada la comunicación. Aunque esto se ha cambiado en últimas versiones del protocolo, permitiendo las conexiones persistentes [7].

Por otro lado, la arquitectura de los servicios web está muy orientada al esquema cliente-servidor, esto es así porque en una aplicación compuesta de *webservices*, algunos nodos (servidores) ofrecen las capacidades o los recursos y otros nodos (clientes) son los que las consumen. Esto es así porque la concepción de esta arquitectura ha sido siempre la ofrecer de manera escalable servicios a los usuarios finales. Es por ello una de las arquitecturas más utilizadas en aplicaciones web tanto si son destinadas al usuario final o a la red empresarial.

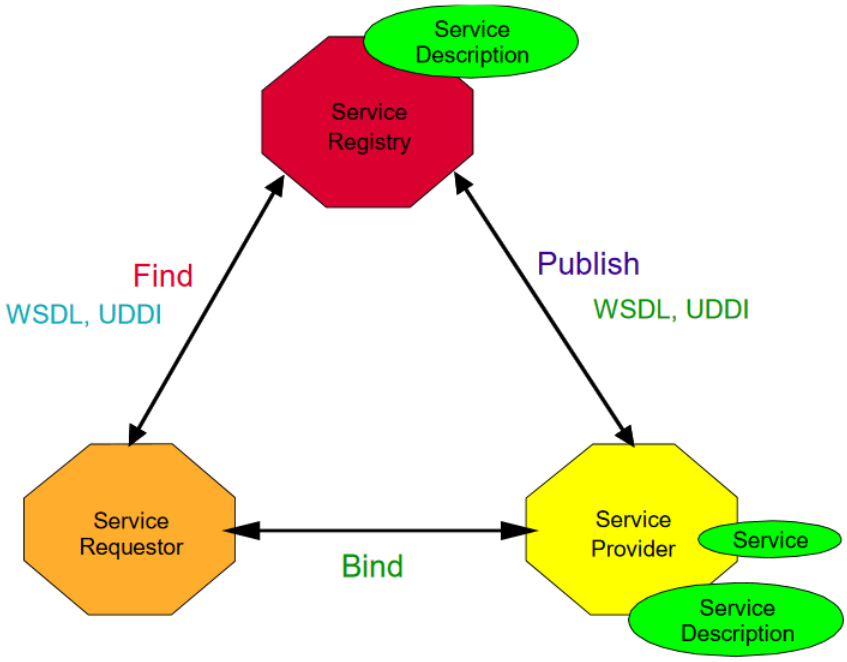


Ilustración 2: Ejemplo de esquema de servicio web

En la Ilustración 2 se puede ver cómo los componentes básicos de un servicio web son: el servidor de registro, el proveedor del servicio y solicitante (consumidor) del mismo. [8] Para consumir el servicio se debe acudir al servicio de registro, obtener la descripción del mismo, acudir a proveedor y, con las credenciales anteriores,

acceder al servicio web. Como se puede ver, es una arquitectura eminentemente ideada con un esquema cliente-servidor.

En un esquema en que los nodos de un sistema de memoria distribuida colaboren en una situación de igualdad (es decir, en una arquitectura plana), los nodos tendrían constantemente que ocupar el rol de solicitante del servicio o de proveedor del mismo sin ningún criterio, lo que haría que el concepto de «servicio» dejara de tener tanto sentido.

La arquitectura RPC (*Remote procedure call*) es, en cierto modo, una versión menos general que la de un servicio web, si bien también tiene un marcado carácter cliente-servidor, utiliza un protocolo propio que permite serializar los datos en modo binario (no en modo texto) y que, por tanto, añade menos sobrecarga al funcionamiento del sistema. El principal problema es que RPC está centrado en los procedimientos. La filosofía de un esquema RPC es que un programa pueda incluir rutinas o funciones cuya ejecución se dé en otra máquina de una manera transparente, y un *framework* se encargue de serializar los datos, de mandarlos y de gestionar la sincronización.

Al tener una arquitectura cliente-servidor tan marcada, debe definirse el comportamiento de ambos participantes si hay un error de conexión. Las semánticas que se utilizan al lanzar procedimientos remotos son:

- Semántica «tal vez»: Se manda el comando de ejecución del procedimiento y no se espera respuesta.
- Semántica «al menos una vez»: Se manda el comando de ejecución del procedimiento y se espera respuesta, si no llega confirmación, se manda una retransmisión, pero el servidor **no** filtra las peticiones duplicadas.
- Semántica de «máximo una vez»: Se manda la orden de ejecución del comando y si en un tiempo dado no llega la respuesta se retransmite el comando, pero el servidor sí es capaz de filtrar las peticiones.
- Semántica de «exactamente una vez»: En este caso, sin importar lo que pase en el servidor, la operación se ejecutará una vez y se confirmará al solicitante.

Con estas condiciones, es evidente que para un entorno como éste lo más deseable es la última, que es la más difícil de lograr. La filosofía de invocar a un procedimiento remoto es ciertamente adecuada para la computación distribuida. En la literatura de puede ver que se han hecho varios de trabajos sobre ese supuesto **[9][18]**.

Pero el principal problema de RPC es que está enfocado a los procedimientos, sería la versión estructurada de la computación distribuida, puesto que no se centra en los objetos de acceso (como sí hacen los servicios web REST). Sin embargo, sí que se podrían implementar accesos a objetos mediante procedimientos. Poniendo en contraste los servicios web con las RPC, para acceder a un objeto en un servicio web se puede acceder a una URL que lo localice (www.servicios.com/libros/293) y según el método de HTTP escogido, realizar una acción u otra. Sin embargo, en RPC se invocaría a un procedimiento (función) para acceder al objeto, por ejemplo «libros_get(293)». Esto provoca un acceso a los datos menos parecido al paradigma de orientación a objetos (centrado en los datos) y más parecido a lo que se haría en programación estructurada (separando datos y procedimientos).

Finalmente, queda la alternativa de bibliotecas de pasos de mensajes como PVM o MPI (*message passing interface*). PVM es una opción obsoleta, por lo que MPI es una

solución ampliamente utilizada que se centra en el paso de mensajes, como su propio nombre indica. Así, las operaciones principales son enviar y recibir datos y utilizar estas llamadas (bloqueantes) como manera de sincronizar los procesos. Esto permite, a su vez, centrarse en los datos, no en las operaciones, a diferencia de las dos soluciones anteriores (RPC y servicios web SOAP) y, por otro lado, es eficiente, utilizando un estilo de comunicación más cercano al protocolo de transporte (TCP) sin tener formatos de texto intermediarios (como XML o JSON) ni protocolos de aplicación como HTTP.

Además, MPI tiene la ventaja de que, al igual que los demás, incluye un sistema de registro automático, dado que una vez se compila el programa, al ejecutarlo los procesos pueden saber su identidad en el clúster (denominado identificador) y cuántos procesos hay en el mismo **[10]**. Esto permite que, mediante el modelado de comportamiento diferentes en el código de cada proceso, ejecutando el mismo programa, puedan colaborar desarrollando diferentes partes del trabajo.

Esto elimina la concepción cliente-servidor que las demás filosofías tienen, puesto que no hay un proveedor de servicio y un consumidor, sino varios procesos que ejecutan el mismo *software* pero que según su identidad en el clúster se comportan según les corresponda en el reparto de tareas.

El patrón para diseñar aplicaciones con MPI es, por tanto, que se cree un código único y se añadan bifurcaciones según el ID del proceso cuando llegue a esas rutinas, lo que permite un esquema similar al multiproceso en una sola máquina. Si en una máquina normal, el proceso podría ser:

1. Iniciar el programa
2. Crear un proceso hijo con *fork*.
3. Si el PID no es 0, soy el padre, luego mando datos al proceso hijo.
4. Si el PID es 0, soy el hijo, luego proceso datos y mando respuesta.
5. FIN.

En un programa en MPI el proceso es similar

1. Lanzamiento de aplicación multiproceso con MPI.
2. Utilizar la interfaz para saber mi identificador.
3. Si mi identificador es 0, mando datos al siguiente.
4. Si mi identificador no es 0, proceso los datos.
5. FIN

En estos ejemplos sólo se lanzan dos procesos, pero la filosofía es extensible a cualquier número de ellos. Esto hace que si lo que se quiere es un reparto de tareas dinámico (que un proceso tenga un rol diferente ante la misma operación según el momento) o simplemente se desea tener una identificación fácil y clara de los procesos, el paso de mensajes sea una buena alternativa.

Por otro lado, MPI se encarga de la semántica de las operaciones que realiza, a diferencia de las RPC, se garantiza la entrega de los mensajes y se dispone de versiones bloqueantes o no de las primitivas enviar o recibir, amén de que se ofrece una colección de llamadas que permiten, no sólo la comunicación y la identificación de los procesos, sino que dan características como manejo concurrente de ficheros,

abstracciones para el manejo de archivos (tales como tipos de datos derivados) y aplicación conjunta de algoritmos, como por ejemplo la reducción de un vector. [11]

Todas estas características hacen a MPI una solución muy adecuada para los propósitos de un sistema de computación distribuida en que los procesos intercambien información y cambien de rol muy rápidamente. Si los procesos no tienen un marcado carácter cliente-servidor, es una gran opción por su sistema de identificación genérico.

Finalmente, a la hora de dotar a C++ de cualidades para la memoria distribuida, el enfoque siempre tiene a ser «apoyarse en hombros de gigantes» y utilizar herramientas de las anteriormente mencionadas para dotar de una interfaz al lenguaje que sea apropiada para estos trabajos. Además, la solución más utilizada en estos casos en MPI, puesto que un lenguaje orientado a objetos está más centrado en los datos que en los procedimientos. Y como hemos visto, el paso de mensajes es una filosofía que sigue este principio.

Por ello, hay trabajos que intentan adaptar problemas de alto coste computacional a un paradigma de memoria distribuida en C++, éste es el caso de *libMesh* [12]. En este trabajo, se expone la creación de una biblioteca que permita cálculos con elementos finitos en memoria distribuida en C++. Además, utiliza también MPI.

Para sus cálculos utilizan una serie de estructuras de datos propias, que se mapean con los conceptos del problema, tales como malla, grados de libertad, nodos, elementos y demás conceptos geométricos que permitan trabajar con un conjunto de puntos conectados. En este caso, para resolución de los problemas que estos elementos tienen, se utiliza la interfaz de paso de mensajes para aprovechar el paralelismo de los procesos matemáticos que en dicha resolución se dan, es decir, se aprovechan a nivel del álgebra lineal. Así, utilizando algoritmos bien estudiados, pueden aprovechar el paralelismo que éstos tienen.

El principal problema de esta interfaz es que es muy específica y, además, no aprovecha de un modo nativo los contenedores de C++. Lo que sí hace es aislar al usuario de la complejidad de utilizar programación distribuida (o simplemente paralela) para que sólo deba concentrarse en el modelo y las operaciones que está realizando. Para esto, realizan sincronizaciones en momentos concretos de los procesos que su biblioteca contempla, utilizando para ellos las primitivas de MPI. Esto coincide en la filosofía de este trabajo: aislar al usuario lo más posible del hecho de que está ejecutando su código en un clúster o en varios procesadores.

Finalmente, en *libmesh*, pese a tener como objetivo desligar al usuario del control de la programación distribuida, la complejidad de los problemas abordados ha hecho necesario que los usuarios de la biblioteca deban proporcionar código cuando se tratan las matrices en paralelo para ciertos procesos. Esto indica que en este trabajo la resolución del problema en paralelo requiere más trabajo de desarrollo que en el caso de operaciones simples.

Otra de las aproximaciones que existen para dotar a C++ de una manera de trabajar con memoria distribuida con MPI aprovechando las cualidades del lenguaje. Debido a que MPI debe ser interoperable en C, C++ y FORTRAN, generalmente su interfaz tiende a ser el menor denominador común de todas ellas, como se indica en el trabajo de P. Kambadur. [13]

MPI tiene una interfaz marcadamente dependiente del estilo de programación típico de C, y para poder aprovechar las características de C++, se deben programar rutinas intermedias que permitan traducir conceptos del segundo al primero. Por ejemplo, el soporte de referencias, la deducción de tipos mediante plantillas o el uso de iteradores y contenedores. Por esto, el enfoque de ese trabajo sería proponer una interfaz a MPI que aprovechase estas características del lenguaje. Esto aportaría la filosofía principal de que es necesario que una biblioteca como MPI, o una que a su vez la utilice aproveche las capacidades de un lenguaje como C++.

Esta interfaz hace que utilizar directamente la biblioteca para trabajos de memoria distribuida sea complejo: se deben traducir conceptos de C++ a otros de C constantemente (referencias a punteros, contenedores a arrays reservados dinámicamente, proporcionar los tipos de dato en vez de usar la deducción de plantillas etc.). Es cierto que sería muy cómodo si la misma interfaz de la librería cambiara a para adaptarse a las nuevas características del mismo. Esto solucionaría muchas de las dificultades, pero no aísla al usuario de los problemas de la programación en memoria distribuida.

Si ya tenemos dos de las piezas que más nos han interesado en este proyecto (aislar al usuario de la biblioteca de la programación concurrente y aprovechar las características de C++), trabajos anteriores aportan el tercer punto de apoyo, que es, además de aislar al usuario de la propia programación concurrente y adaptar los conceptos de C++ a la memoria distribuida con MPI, adaptar también la filosofía de la STL a dicho paradigma.

Para aunar todos esos conceptos hay un trabajo anterior **[14]** que usa la misma filosofía que el trabajo que *libMesh* a la hora de aislar al usuario de la computación en memoria distribuida, pero, además, implementa sus propias soluciones que respetan e integran el modelo de C++ basado en iteradores y algoritmos. Esto quiere decir que el objetivo es que, utilizando contenedores, iteradores y algoritmos con las mismas interfaces que los de la STL el usuario trabaje sin intervenir en operaciones de paso de mensajes. Su intención es que, simplemente sustituyendo los contenedores y los algoritmos por los de su propia librería, se ejecuten paralelamente.

Como se puede ver, la mayoría de soluciones que desean dotar a C++ de una interfaz diáfana para trabajar con memoria distribuida tienden a adaptar las poderosas herramientas del lenguaje a las herramientas que utilicen por su parte. Además, el paso de mensajes suele ser el paradigma de computación distribuida elegido porque se centra en los datos, y no en las operaciones, lo que encaja con la programación orientada a objetos y con el concepto de contenedor.

En resumen, las limitaciones que C++ ha tenido con la concurrencia en su concepción inicial obligaron a que las implementaciones de la misma en él se añadieran como bibliotecas estandarizadas de terceros. Además, la herramienta tecnológica más adecuada para lograrlo ha sido tradicionalmente el paso de mensajes. Esto es porque, en la mayoría de casos, cuando se programa en un paradigma de orientación a objetos se traslada el protagonismo del *software* a los datos, en vez de a los procedimientos, como sí pasaría en programación estructurada. Esta característica no está disponible en C++ *per se*, pero sí con alternativas como MPI.

Y, finalmente, el estándar *de facto* para el uso de paso de mensajes viene siendo durante años MPI, que tiene una interfaz dependiente de las particularidades de C. Por todo esto, en este trabajo vamos a buscar como objetivo principal compatibilizar los conceptos de C++ con la interfaz de MPI y además aislar lo más posible -si no del todo- al programador de los problemas del uso de memoria distribuida.

4. Entorno de desarrollo

En la sección anterior se han discutido las alternativas para el desarrollo de la aplicación propuesta, si bien ahora se describirán los detalles del software y hardware escogidos para el desarrollo y evaluación de la misma. Por otro lado, se explicarán las herramientas utilizadas en el desarrollo.

4.1 MPICH

MPI tiene varias implementaciones, debido a que es un proyecto de código libre, por ello, varias entidades tienen sus propias implementaciones y hay dos que destacan sobre el resto prominentemente:

- MPICH
- OpenMPI

No hay una razón fundamental para decantarse por ninguna de las dos implementaciones, pero MPICH ha sufrido una gran mejora de rendimiento en la versión 2 y 3 y, además, hace uso de algoritmos más sofisticados para la realización de operaciones colectivas **[30]**, de tal modo que es una opción en aplicaciones más complejas. Por otro lado, muchas variantes creadas por empresas son compatibles en MPICH, lo que permite su compatibilidad con ellas mejor que otras alternativas **[31]**. Por añadidura, el grupo de investigación ARCOS tiene una estrecha línea de colaboración con *Argonne National Laboratory* (Chicago, EE.UU.), que es uno de los principales desarrolladores de MPI, lo cual apoya también la elección.

4.2 C++14

El estándar de C++ publicado en 2014 es el que estaba publicado y consolidado cuando se propuso este proyecto y, por otro lado, incluye novedades importantes que han sido de gran ayuda en la implementación del *software*.

- Mejoras en las expresiones lambda.
- Utilización de la palabra clave «auto» para declaración de variables.
- Separador de dígitos para los números con apóstrofo (`1'000'000`).

4.3 Herramientas de desarrollo

El entorno de desarrollo estuvo compuesto de un PC ejecutando Windows 10 y con la *bash* de Linux, tal como ofrece dicha versión del sistema operativo de Microsoft **[32]**. Esto ha permitido la compilación y ejecución de las diferentes versiones de prueba del *software* de manera más rápida, a la vez que su documentación.

Para la edición del código de desarrollo se ha utilizado el editor de texto Sublime Text. Un editor de código utilizado ampliamente en entornos de desarrollo en lenguajes como C++.

Para la compilación del *software* se ha utilizado el compilador GNU para C++ en su versión 6.2. Junto con las herramientas *Cmake* y *make* para la gestión de los de la compilación.

4.4 Entorno *hardware*

El entorno hardware utilizado incluye:

- Equipo de desarrollo:
  - Ordenador doméstico, características:
    - Procesador: Intel Core i7-2600. 8 núcleos a 3,7 GHz, *multithreading*.
    - 8 GB memoria RAM DDR3-1333 Hz.
    - 500 GB de disco duro en 1 disco.
  - Hardware de pruebas:
    - Clúster Tucan de la universidad Carlos III de Madrid, equipos utilizados:
      - Nodos Compute 1:
        - Procesador: Intel Xeon E5405 @ 2 GHz
          8 núcleos
      - Nodos compute 3:
        - Procesador: Intel Xeon E5405 @ 2GHz
          8 núcleos
      - Nodos compute 6:
        - Procesador: Intel Xeon E5645 @ 2.4 GHz
          24 núcleos
      - Nodos Compute 11:
        - Procesador: Intel Xeon E5-2603 v4 @ 1.70 GHz
          – 12 núcleos

4.5 Marco regulador

En este trabajo se han utilizado fundamentalmente dos estándares técnicos:

- MPI
- C++ 14.

MPI es un estándar desarrollado en los años 90 y que ha pasado por sucesivas versiones hasta la actualidad. Está ideado como una manera de permitir el paso de mensajes y añade, además, una colección de operaciones de entrada y salida paralela. Además, es la tecnología de facto en este terreno. No está ligada a un compilador ni a una implementación concreta y permite ser utilizado en varios lenguajes. Los más utilizados con él son C/C++ y Fortran. Está publicado por el MPI Forum, que es el foro de estandarización. La implementación elegida (MPICH) es de código libre y dominio público. Utiliza una licencia de código libre que permite el uso, reproducción, utilización del trabajo como base para otros y distribuirlo a terceros.

C++ 14 es el último estándar liberado del lenguaje de programación C++ cuando se empezó este desarrollo. Está publicado por la ANSI.

Debido a que todos los componentes *software* son de código libre bajo licencia GNU o de dominio público, no se requeriría gestión de dichas licencias para una explotación comercial del *software*. Además, el objetivo es distribuir el software con una licencia GPL para su uso por parte de la comunidad de C++.

5. Descripción de la biblioteca

En esta sección se va a describir el diseño del componente mediante una visión de 5 vistas UML (lógica, desarrollo, proceso y física) además de describir la metodología de trabajo y los objetivos principales de la biblioteca. Además, se hará un seguimiento de la trazabilidad con el fin de exponer que todas las decisiones de diseño tienen su raíz en los requisitos y objetivos iniciales de la biblioteca.

El objetivo de la misma es dotar a C++ de una biblioteca que permita la ejecución de programas en C++ memoria distribuida con el principal contenedor proporcionado en la STL, que es el vector. Para esto se desarrollará una biblioteca que permita, usando una interfaz igual a la de la misma STL el trabajo con este contenedor nuevo y los iteradores de la misma. Además, se aísla al usuario (programador) de las particularidades de la programación para entornos de memoria distribuida.

5.1 Metodología de trabajo

La metodología de trabajo ha seguido el siguiente proceso:

1. Establecimiento de la idea principal.
2. Establecimiento de requisitos de usuario.
3. Diseño basado en requisitos.
4. Implementación.
5. Validación del diseño derivado de los requisitos.
6. Repetición iterativa de los pasos 2, 3, 4 y 5.

De este modo, el proyecto se planteó con una idea inicial que, después, fue llevada a cabo añadiendo los requisitos propios del *software* que se desea perfilar, se procedía a un diseño para la implementación de esos requisitos y su implementación, se validaban el diseño y la implementación y se repetía el proceso de licitación de requisitos, diseño, implementación y validación hasta terminar el desarrollo del software.

A continuación de describe el ciclo de desarrollo, donde una casilla verde significa que ese día fue dedicado a esa tarea.

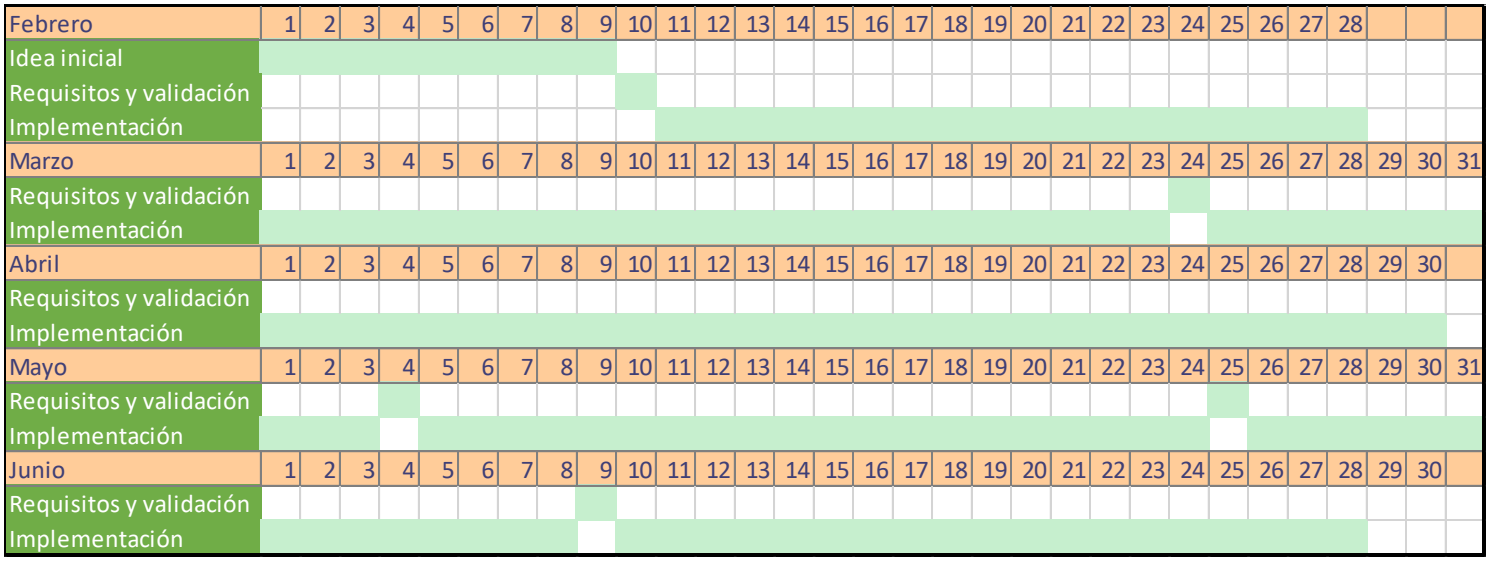


Ilustración 3: Diagrama de Gantt de la fase de desarrollo

En paralelo con la fase de desarrollo, se pasó a la fase de documentación y elaboración de la memoria, que constó de las siguientes fases:

Cambios	Fecha Inicio
Primera escritura del estado de la cuestión	29/5/2017
Escritura de los requisitos de usuario y de sistema	1/6/2017
Revisión del estado de la cuestión.	2/6/2017
Requisitos y casos de uso	04/06/2017
Escritura del diseño	06/06/2017
Pruebas de validación	08/06/2017
Revisión de requisitos y casos de uso	10/06/2017
Escritura de la Introducción a C++ y MPI. Revisión de diseño detallado.	12/06/2017
Revisión introducción a C++ y MPI y escritura pruebas de rendimiento.	13/06/2017
Escritura introducción y conclusiones	18/06/2017
Última versión del documento	20/06/2017

Tabla 2: Planificación de la escritura de la memoria.

Dichas fases de detallan en la **Ilustración 4**: Diagrama de Gantt de la escritura de la memoria:

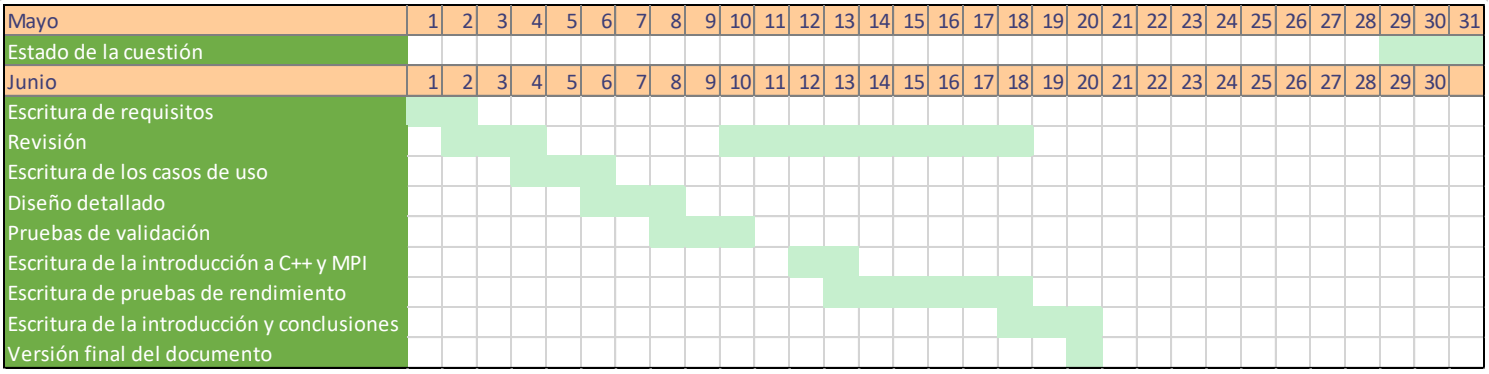


Ilustración 4: Diagrama de Gantt de la escritura de la memoria

5.2 Requisitos

En esta sección se van a desglosar los requisitos de usuario y los de sistema, el esquema de cada requisito será una tabla como la siguiente:

REQ-XX-YY	
Título	
Descripción	
Requisitos relacionados	

Tabla 3: Tabla tipo para los requisitos

- El título de la tabla es un identificador con la estructura REQ-XX-YY donde XX es el tipo de requisito (US para usuario y SI para sistema) e YY son dos cifras que identifican al requisito. Podrá haber dos requisitos de tipo distinto con mismo número, tal como «REQ-US-20» y «REQ-SI-20».
- Título: frase concisa que pueda dar una idea del contenido del requisito.
- Descripción: Texto del requisito, especificación del mismo.
- Los requisitos relacionados indican de qué otros requisitos deriva el descrito o con qué requisitos de usuario se relacionan los de sistema. (Esta información se incluirá luego en una matriz de trazabilidad)

5.2.1 Requisitos de usuario

Los requisitos de usuario serían:

REQ-US-01	
Título	Contenedor vector
Descripción	La biblioteca deberá proporcionar una versión distribuida del contenedor `std::vector` en la que los elementos del mismo serán distribuidos según modos de reparto establecidos en requisitos posteriores entre los procesos intervinientes en el sistema.
Requisitos relacionados	Ninguno

REQ-US-01: Contenedor vector

REQ-US-02	
Título	Modos de reparto
Descripción	El componente proporcionará al usuario un mecanismo para que pueda elegir qué modo de reparto se elige para cada vector.
Requisitos relacionados	REQ-US-01

REQ-US-02: Modos de reparto

REQ-US-03	
Título	Catálogo de modos de reparto
Descripción	Los modos de reparto disponibles serán: • Bloque. • Round Robin de rodaja variable. • *Ad-hoc*. • Optimizado
Requisitos relacionados	REQ-US-02

REQ-US-03: Catálogo de modos de reparto

REQ-US-04	
Título	Reparto de bloque
Descripción	El reparto de bloque se corresponde a su especificación formal: **[Especificación formal de los modos de reparto]**
Requisitos relacionados	REQ-US-03

REQ-US-04: Reparto de bloque

REQ-US-05	
Título	Reparto tipo Round Robin
Descripción	El reparto de bloque se corresponde a su especificación formal: **[Especificación formal de los modos de reparto]**
Requisitos relacionados	REQ-US-03

REQ-US-05: Reparto tipo Round Robin

REQ-US-06	
Título	Tipo de reparto *ad-hoc*
Descripción	El reparto de bloque se corresponde a su especificación formal: **[Especificación formal de los modos de reparto]**
Requisitos relacionados	REQ-US-03

REQ-US-06: Tipo de reparto ad-hoc

REQ-US-07	
Título	Tipo de reparto optimizado
Descripción	El tipo de reparto optimizado será como un tipo *ad-hoc*, pero la longitud de los bloques no será proporcionada por el usuario, sino que será calculada como un reparto inversamente proporcional al tiempo que los procesos tardaran en ejecutar una prueba de rendimiento.
Requisitos relacionados	REQ-US-03

REQ-US-07: Tipo de reparto optimizado

REQ-US-08	
Título	Prueba de rendimiento
Descripción	El sistema evaluará el rendimiento del conjunto de máquinas que ejecuten el programa. Para esto existirá un parámetro controlable por el usuario que indique cuándo se debe hacer esta medición.
Requisitos relacionados	REQ-US-07

REQ-US-08: Prueba de rendimiento

REQ-US-09	
Título	Modos de acceso
Descripción	El componente deberá proporcionar al usuario una interfaz «transparente» que permita las mismas operaciones que con un vector estándar en C++ sin tener que involucrarse en memoria distribuida. Las operaciones serían: • Utilización de operador de acceso (operador[]) **[INTRODUCCIÓN A C++]** • Utilización de iterador de tipo *forward* **[19]** Utilización del iterador con algoritmos de la STL.**[24]**
Requisitos relacionados	Ninguno

REQ-US-09: Modos de acceso

REQ-US-10	
Título	Algoritmos distribuidos
Descripción	Se proporcionará un conjunto de funciones dentro de la librería que sigan la misma interfaz que las de la STL y que distribuyan los cálculos del vector entre los diferentes procesos. Los algoritmos ofrecidos serán: 1. Reduce: Aplica un operador entre todos los elementos del rango al que se aplique. El operador debe ser conmutativo y asociativo. 2. Transform: Aplica una función a cada elemento del rango donde aplica para cambiar los elementos.
Requisitos relacionados	REQ-US-09

REQ-US-10: Algoritmos distribuidos

REQ-US-11	
Título	Tipos de dato del vector
Descripción	El vector deberá poder contener sólo dos tipos de dato: **int** y **double**. Cada vector sólo podrá contener datos de uno de los tipos y esa característica del mismo se decidirá en tiempo de compilación.
Requisitos relacionados	REQ-US-1

REQ-US-11: Tipos de dato del vector

REQ-US-12	
Título	Impresión por salida estándar
Descripción	El componente deberá proporcionar un mecanismo para que si se produce salida por la salida estándar ésta sea reproducida sólo una vez, no una por proceso. De tal modo que: Imprimir(x) produzca esta salida: "x" Y no n impresiones iguales. Siendo n el número de procesos.
Requisitos relacionados	REQ-US-09

REQ-US-12: Impresión por salida estándar

REQ-US-13	
Título	Detección de despliegue en otras máquinas
Descripción	Si el programa se ejecuta después de medir el rendimiento en el mismo subconjunto de máquinas, aunque el orden de las mismas sea diferente, se detectará y se harán los cambios oportunos para la ejecución en modo optimizado.
Requisitos relacionados	REQ-US-08

REQ-US-13: Detección de despliegue en otras máquinas

REQ-US-14	
Título	Lectura de archivo
Descripción	La biblioteca ofrecerá una interfaz para la lectura desde un archivo binario que llene el vector con los datos numéricos que estén en ese archivo. El archivo deberá contener una copia de los *Bytes* en memoria que representaban dichos números. No es un archivo de texto.
Requisitos relacionados	REQ-US-09

REQ-US-14: Lectura de archivo

REQ-US-15	
Título	Escritura del archivo
Descripción	La biblioteca proporcionará una interfaz que permita escribir en un archivo binario el contenido íntegro del vector distribuido o un número de sus elementos (siempre empezando desde el primer elemento). Este fichero tendrá el mismo formato que el que se usa para llenarlo.
Requisitos relacionados	REQ-US-14

REQ-US-15: Escritura del archivo

REQ-US-16	
Título	Interfaz optimizada
Descripción	El sistema ofrecerá una interfaz optimizada. En esta interfaz las expresiones que involucren el operador [] de acceso se comportarán de este modo: v[ii] retornará el valor correcto sólo en el nodo que lo almacena localmente según el tipo de reparto. Esta interfaz será activada mediante un símbolo definido en tiempo de compilación. (por ejemplo, con #define SYMBOL)
Requisitos relacionados	REQ-US-3, REQ-US-4, REQ-US-5, REQ-US-6, REQ-US-7

REQ-US-16: Interfaz optimizada

Especificación formal de los modos de reparto

La definición formal, expresada mediante funciones matemáticas, de los modos de reparto es la que sigue:

1. Reparto de bloque: En este modo de reparto, un elemento i del vector es almacenado en el proceso j de este modo:

   $$V_i \in P_j \Leftrightarrow \left\lfloor \frac{|V|}{|P|} \right\rfloor \cdot j \leq i < \left\lfloor \frac{|V|}{|P|} \right\rfloor \cdot (j+1)$$

   Salvo en el caso del último proceso ($j = |P| - 1$), en que la expresión sería:

   $$V_i \in P_{|P|-1} \Leftrightarrow \left\lfloor \frac{|V|}{|P|} \right\rfloor \cdot (|P| - 1) \leq i$$

   V *es el conjunto de elementos del vector*
   P *es el conjunto de procesos*

2. Reparto cíclico o Round Robin: Este modo de reparto se basa en dividir el vector en rodajas de tamaño n, siendo n elección del usuario, que serán asignadas a los procesos cíclicamente. Si no se pudiera dividir el vector en rodajas de tamaño n, la última rodaja sería de la longitud de elementos que restare para completar el vector. Formalmente:

   $$V_i \in R_j \Leftrightarrow n \cdot j \leq i < n \cdot (j+1)$$
   $$R_j \in P_k \Leftrightarrow j \equiv k \; mod(|P|)$$

   V *es el conjunto de elementos del vector*
   P *es el conjunto de procesos*
   R *es el conjunto de rodajas en que se parte el vector*

3. Reparto *ad-hoc*: En este tipo de reparto el usuario indica las longitudes de los bloques que le corresponden a cada proceso. Es decir, sea el conjunto de longitudes L

   Si $j \neq 0$ entonces:

   $$V_i \in P_j \Leftrightarrow \sum_{k=0}^{j-1} l_k \leq i < \sum_{k=0}^{j} l_k$$

   Si $j = 0$ entonces:

   $$0 \leq i < l_0$$

5.2.2 Requisitos de sistema

REQ-SI-01	
Título	Clases expuestas
Descripción	La biblioteca deberá exponer las siguientes clases para su uso por parte del usuario: 1. dcpl::DistributedVector 2. dcpl::DistributedVector::iterator 3. dcpl::ifstream 4. dcpl::inicializador Ver: **DISEÑO LÓGICO**
Requisitos relacionados	REQ-US-01, REQ-US-09, REQ-US-14, REQ-US-15, REQ-US-16

REQ-SI-01: Clases expuestas

REQ-SI-02	
Título	Instanciación de la clase DistributedVector
Descripción	La clase deberá ser pública y dispondrá de uno o varios constructores públicos que permitan proporcionar la información del reparto de elementos. Dicha información será: 1. Tipo de reparto, que será de los siguientes valores: BLOQUE, ROUND ROBIN, AD-HOC, BENCHMARK u OPTIMIZED. 2. Amplitud de la rodaja, sólo solicitado si el tipo es Round Robin. La amplitud de la rodaja será un número entero no negativo ni nulo. 3. Longitudes de los bloques que componen el reparto *ah-hoc*. Sólo utilizado si se elige ese tipo de reparto. Las longitudes de los bloques serán una lista de números enteros estrictamente mayores que cero. Además, dicha lista deberá contener tantos elementos como procesos se lancen.
Requisitos relacionados	REQ-SI-01, REQ-US-02, REQ-US-03, REQ-US-08

REQ-SI-02: Instanciación de la clase DistributedVector

REQ-SI-03	
Título	Control de contexto
Descripción	La biblioteca guardará la información relativa a MPI en una estructura de datos. Esta información será: 1. Número de procesos 2. Rango (identificador) del proceso.
Requisitos relacionados	—

REQ-SI-03: Control de contexto

REQ-SI-04	
Título	Obtención de tiempos de ejecución
Descripción	Si se instancia un objeto o más de la clase DistributedVector con el tipo de reparto BENCHMARK, se guardarán los tiempos de ejecución del programa en un fichero. Se registrarán en milisegundos. Además, el fichero tendrá el formato: Tiempo 1 | MD5(nombre_nodo1) … | … Tiempo n | MD5(nombre_nodo_n) Donde los tiempos son escritos con los 4 B que los conforman en memoria principal y MD5(nombre) como el resultado de aplicar la función MD5 a la salida del comando «uname -n» para ese nodo. El resumen medirá 16 *Bytes*.
Requisitos relacionados	REQ-US-02, REQ-US-07, REQ-US-08, REQ-US-13

REQ-SI-04: Obtención de tiempos de ejecución

REQ-SI-05	
Título	Recuperación de tiempos de ejecución
Descripción	Si el tipo de reparto escogido para algún vector es OPTIMIZED, entonces la biblioteca recuperará del archivo previamente escrito los tiempos de ejecución del modo BENCHMARK. Si no existiera el archivo, se crearía un vector con modo de reparto en BLOQUE en su lugar.
Requisitos relacionados	REQ-SI-04, REQ-US-08

REQ-SI-05: Recuperación de tiempos de ejecución

REQ-SI-06	
Título	Tecnología de manejo de archivo de datos
Descripción	Para la lectura y la escritura del archivo que llenará el vector se deben utilizar *datatypes* derivados ofrecidos por MPI. Se elaborará un tipo de dato que se corresponderá con los elementos que pertenecen a ese proceso (ver requisitos REQ-US-04, REQ-US-05, REQ-US-06 y REQ-US-07)
Requisitos relacionados	REQ-US-04, REQ-US-05, REQ-US-06, REQ-US-07, REQ-US-14, REQ-US-15

REQ-SI-06: Tecnología de manejo de archivo de datos

REQ-SI-07	
Título	Contenido de la clase DistributedVector
Descripción	La clase guardará la siguiente información: 1. El esquema de reparto. 2. Std::vector con los datos del DistributedVector que le correspondan a este proceso. *3.* *Datatype* derivado de MPI que se utilice para leer el archivo.
Requisitos relacionados	REQ-US-01, REQ-US-02, REQ-US-14, REQ-US-15

REQ-SI-07: Contenido de la clase DistributedVector

REQ-SI-08	
Título	Interfaz de la clase DistributedVector
Descripción	La clase expondrá los siguientes métodos públicos: 1. Operador [] (int pos): Proporcionará acceso al elemento en la posición pos. En modo normal se comportará acorde a REQ-US-09. En modo optimizado su comportamiento viene descrito en REQ-US-16. 2. Get (**pos**, **nodo**): devolverá el valor de la posición **pos**. Pero sólo en el **nodo** indicado. En los demás devolverá 0. Si **nodo** es menor que 0, se devolverá el dato correcto en el nodo que lo aloje, y en el resto, 0. 3. Set (pos, valor): Hará que el proceso que aloje el elemento en **pos** cambie su contenido a **valor**. El resto de procesos no hará nada. 4. Size: Devolverá el número de elementos en el vector distribuido. 5. Begin: devolverá un dcpl::Distributedvector::iterator que apunte al primer elemento del vector. 6. End: Devolverá un dcpl::Distributedvector::iterator que apunte al elemento siguiente al último elemento del vector.
Requisitos relacionados	REQ-US-01, REQ-US-09

REQ-SI-08: Interfaz de la clase DistributedVector

REQ-SI-09	
Título	Dcpl::DistributedVector será una plantilla de clase
Descripción	DistributedVector será una plantilla de clase con un argumento de tipo *typename*. Sólo admitirá los tipos int y double. El argumento de la plantilla será el tipo de los elementos que el vector puede contener.
Requisitos relacionados	REQ-US-11

REQ-SI-09: Dcpl::DistributedVector será una plantilla de clase

REQ-SI-10	
Título	Constructor de la clase dcpl::ifstream
Descripción	La clase ifstream dispondrá de un único constructor cuyo argumento único será la ruta al archivo que se quiere manejar: `Ifstream(String path)`
Requisitos relacionados	REQ-SI-1

REQ-SI-10: Constructor de la clase dcpl::ifstream

REQ-SI-11	
Título	Interfaz de la clase `dcpl::ifstream`

Descripción	La clase ifstream dispondrá de dos métodos públicos con la siguiente interfaz:
	`ifstream& read(DistributedVector, int length);`
	`ifstream& write(distributedVector, int length);`
	En el caso de read, se leerán N elementos del archivo binario que se indicó en la instanciación del objeto ifstream para introducirlos como datos del vector según el modo de reparto escogido. Siendo $N = \min(length,\ elementos\ en\ el\ fichero)$
	En el caso de write, se escribirán N elementos del vector en un archivo binario indicado por la ruta que se pasó al instanciar el objeto ifstream. Siendo N= $\min(\lvert V \rvert,\ length)$
	Tanto en lectura como en escritura, se leerán y se escribirán N elementos contando desde el primero.

Requisitos relacionados	REQ-SI-10, REQ-US-14, REQ-US-15

REQ-SI-11: Interfaz de la clase `dcpl::ifstream`

REQ-SI-12	
Título	Clase dcpl::DistributedVector::iterator
Descripción	Se creará como *inner class* de DistributedVector la clase Iterator.
Requisitos relacionados	REQ-US-09

REQ-SI-12: Clase dcpl::DistributedVector::iterator

REQ-SI-13	
Título	Tipo de dcpl::DistributedVector::iterator
Descripción	El iterador será de tipo *forward* **[19]**.
Requisitos relacionados	REQ-US-09, REQ-SI-12, REQ-SI-01

REQ-SI-13: Tipo de dcpl::DistributedVector::iterator

REQ-SI-14	
Título	Comportamiento del iterador
Descripción	Expresiones aceptadas por un iterador del tipo *forward* y que, por tanto, `dcpl::DistributedVector::iterator` admitirá, sea el iterador a y el iterador b: 1. Desreferenciación: *a. Devuelve el elemento apuntado por a. 2. Preincremento: ++a. Devuelve el iterador a la posición siguiente en la que estaba antes de ejecutar la operación. Además, modifica el iterador. 3. Postincremento: a++. Incrementa el elemento al igual que la operación anterior, pero devuelve el valor original. 4. Comparación: a==b. Comprueba que los dos elementos apunten a la misma posición del mismo vector.
Requisitos relacionados	REQ-SI-01, REQ-US-09, REQ-SI-12

REQ-SI-14: Comportamiento del iterador

REQ-SI-15	
Título	Compatibilidad del iterador con STL
Descripción	El iterador deberá ser compatible con todos los algoritmos de la STL que lo sean con `std::vector::iterator`. Por esto, deberá definir los siguientes tipos: 1. `Dcpl::DistributedVector::iterator::value_type`: Será el tipo int o double dependiente del argumento de la plantilla al crear el DistributedVector. 2. `Dcpl::DistributedVector::iterator::reference`: Será el tipo "referencia" al tipo anterior (int& o double&). 3. `Dcpl::DistributedVector::iterator::pointer`: Será un puntero al tipo de la plantilla, es decir será (int* o double*) 4. `Dcpl::DistributedVector::iterator::iterator_category`: Será `std::forward_iterator_tag`, es requisito para ser un iterador de tipo *forward*.
Requisitos relacionados	REQ-SI-09, REQ-SI-13

REQ-SI-15: Compatibilidad del iterador con STL

REQ-SI-16	
Título	Interfaz del algoritmo reduce
Descripción	Se expondrá un método público dentro del espacio de nombres (*namespace*) dcpl que tenga la siguiente interfaz: `Reduce(DistributedVector::iterator first, DistributedVector::Iterator last, U init, BinaryOperator op);` Siendo U el tipo int o double. U podrá ser el mismo tipo que el tipo de los elementos del vector correspondiente a los iterador *first* y *last* o no. BinaryOperator será un operador binario que tiene que tener las siguientes características: 1. Su interfaz será: `T op(T a, T b)`: siendo T el tipo *int* o *double*. 2. Deberá ser conmutativo: $op(a,b) = op(b,a)$ 3. Deberá ser asociativo: $op(op(a,b),c) = op\big(a, op(b,c)\big)$ Cualquier tipo invocable será válido como operador binario si cumple lo anterior. Estos son exactamente: Funciones lambda, objetos con operador paréntesis sobrecargado y punteros a función. Ver: **[INTRODUCCIÓN A C++]** En caso de que sea un objeto y el operador paréntesis acceda a variables de clase o si es una función lambda que captura variables el comportamiento será **no definido**.
Requisitos relacionados	REQ-US-10

REQ-SI-16: Interfaz del algoritmo reduce

REQ-SI-17	
Título	Comportamiento del algoritmo reduce
Descripción	El método reduce con la interfaz descrita en REQ-SI-16 tendrá el siguiente comportamiento: Considerando los elementos entre *first* y *last* como un vector V de n elementos: $reduce(first, last, init, op) = a_{n+1}$ $a_i = op(a_{i-1}, v_i), a_0 = init$
Requisitos relacionados	REQ-US-10, REQ-SI-16

REQ-SI-17: Comportamiento del algoritmo reduce

REQ-SI-18	
Título	Interfaz algoritmo transform
Descripción	Es *namespace* dcpl expondrá una función pública con la siguiente interfaz: `Transform(DistributedVector::iterator first, DistributedVector::iterator last, DistributedVector::iterator result, unaryOperator op);` *Result* debe apuntar a un elemento de un vector que permita insertar los datos contenido en el intervalo [*first*, *last*). Además, el prototipo de `op` será: T (U element); U deberá ser del mismo tipo que los elementos del vector al que apunten *first* y *last* o uno convertible implícitamente a él. T será del mismo tipo que los elementos del vector al que apunte *result* o uno implícitamente convertible a él. `Op` podrá ser de uno de los siguientes tipos: funciones lambda, objetos con operador paréntesis sobrecargado y punteros a función. El prototipo de invocación de `op` deberá ser `T (U elemento);` Siendo `U` el tipo de dato del vector al que referencian `first` y `last` o un tipo implícitamente convertible y `T` el tipo de dato del vector al que apunta `result` o uno implícitamente convertible. En caso de que sea un objeto y el operador paréntesis acceda a variables de clase o si es una función lambda que captura variables el comportamiento será **no definido**.
Requisitos relacionados	REQ-US-10

REQ-SI-18: Interfaz algoritmo transform

REQ-SI-19	
Título	Instanciación del inicializador
Descripción	Para que la biblioteca funcione correctamente se debe instanciar un objeto que la clase `dcpl::inicializador`, esta clase dispondrá de un constructor público con el siguiente prototipo: `Inicializador(int a, char** b).` Además, cuando se instancia, se le deben pasar como argumento `argc` y `argv`.
Requisitos relacionados	—

REQ-SI-19: Instanciación del inicializador

REQ-SI-20	
Título	Recopilación de datos del contexto
Descripción	En el constructor del inicializador se recopilarán los datos del contexto. (REQ-SI-03) Estos son: el identificador del proceso, el número de procesos ejecutando ese código y los tiempos de ejecución de inicio del programa y de fin del mismo.
Requisitos relacionados	REQ-SI-04

REQ-SI-20: Recopilación de datos del contexto

REQ-SI-21	
Título	Procedimiento de destrucción del inicializador
Descripción	En el destructor de la clase inicializador se deberán realizar las siguientes tareas: 1. Si no se deben guardar los tiempos en un fichero: o (ver REQ-SI-04) se invoca a MPI_Finalize y se termina el destructor. 2. En caso contrario: o Se deben conseguir los nombres de todos los nodos y hacerse su resumen con el algoritmo MD5. o Se comprueba si el fichero de tiempos existe y, si es así, se elimina. o Se guardan los tiempos en el fichero. o Se llama a MPI_Finalize.
Requisitos relacionados	REQ-SI-04

REQ-SI-21: Procedimiento de destrucción del inicializador

REQ-SI-22	
Título	Algoritmo de balanceo de carga
Descripción	El reparto de elementos entre los procesos se hace del siguiente modo: Sea el conjunto de tiempos $T = \{t_1, t_2, \dots, t_n\}$, con los tiempos de ejecución de cada proceso, los elementos del vector se repartirán de manera inversamente proporcional al tiempo tardado, formalmente: $$M = \frac{N}{\sum_{i=1}^{n} t_i^{-1}}$$ $$n_i = M \cdot t_i^{-1}$$ Siendo n_i el número de elementos que le corresponden al proceso i.
Requisitos relacionados	REQ-US-07

REQ-SI-22: Algoritmo de balanceo de carga

REQ-SI-23	
Título	Impresión sólo por parte del nodo o proceso 0
Descripción	El namespace tendrá un objeto público del tipo `std::ostream` llamado `cout` (es decir, su denominación sería `dcpl::cout`) Este objeto será diferente para el nodo 0 respecto al resto de nodos: 1. Para el 0 será una copia de `std::cout`. 2. Para el resto será un `std::ostream` instanciado de este modo: `ostream(nullptr)` Esto provocará que cuando un nodo imprima por la salida estándar utilizando el objeto `dcpl::cout` sólo produzca resultado si es el nodo 0.
Requisitos relacionados	REQ-US-12

REQ-SI-23: Impresión sólo por parte del nodo o proceso 0

5.2.3 Trazabilidad entre requisitos de usuario y requisitos de sistema

En la siguiente tabla se muestran los requisitos de usuario en las filas y los de sistema en las columnas y se indica con una marca de verificación (✓) si el requisito de sistema de esa columna satisface el requisito de usuario de esa fila.

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
01	✓	✓					✓	✓															
02		✓		✓			✓																
03		✓																					
04					✓	✓																	
05						✓																	
06						✓																	
07				✓		✓																✓	
08		✓		✓	✓																		
09	✓							✓				✓		✓									
10																✓	✓	✓					
11									✓														
12														✓									✓
13				✓																			
14	✓					✓	✓				✓												
15	✓					✓	✓				✓												
16	✓																✓						

Tabla 4: Matriz de trazabilidad entre requisitos de usuario y de sistema

Como se puede ver, en la matriz hay filas con varias marcas de visto y columnas sin ninguna. Esto es porque si bien un requisito de usuario puede ser implementado en múltiples requisitos del sistema, un requisito del sistema puede no estar directamente mapeado con un requisito de usuario.

5.3 Casos de uso

A continuación, se van a describir los casos de uso de la biblioteca, primero haciendo una enumeración de los mismos e indicando la relación de derivación entre ellos y después especificando cada uno. Lo casos de uso contemplados son:

1. Inicialización de la biblioteca
2. Instanciación de un vector distribuido
    a. Con reparto en bloque
    b. Con reparto round Robin
    c. Con reparto *ad-hoc*
    d. Con reparto benchmark
    e. Con reparto optimizado
3. Lectura de un vector desde archivo binario
    a. Lectura de más elementos de los que contiene el vector
    b. Lectura de menos elementos de los que contiene el vector
4. Operaciones con el vector
    a. Reduce
    b. Transform
        i. Caso más simple (optimización)
    c. Usar método get
    d. Usar método set
    e. Operador []
    f. Uso del iterador
        i. Referenciar
        ii. Comparar
        iii. Incrementar
        iv. Postincrementar
        v. Preincrementar
5. Escritura del vector
    a. Mismos subcasos que lectura

En la Ilustración 5 se pueden observar de manera gráfica los casos de uso en su relación con el usuario de la biblioteca y los que, a su vez, son derivados unos de otros.

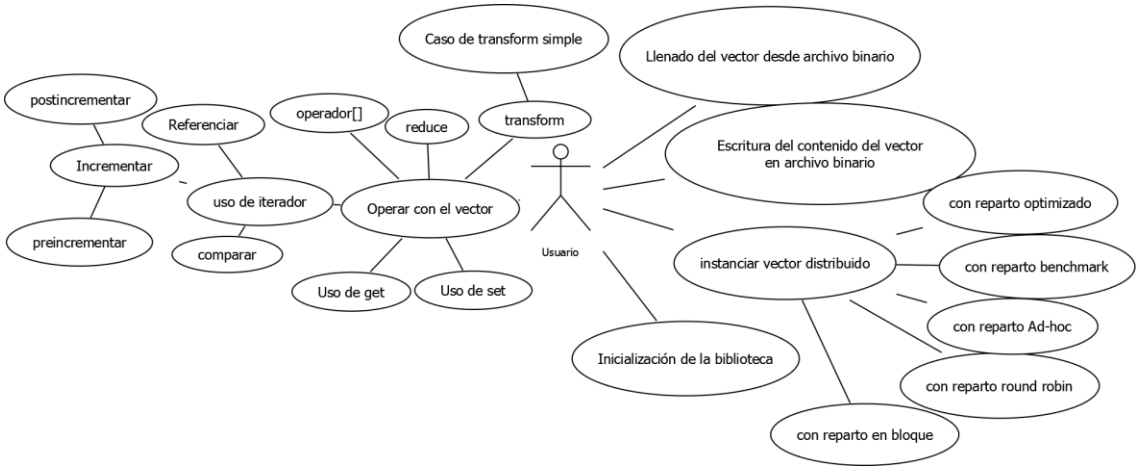


Ilustración 5: Casos de uso principales y derivados

La tabla tipo de la tipificación de un caso de uso será la siguiente:

CU-XX	
Título	
Descripción	
Precondiciones	
Postcondiciones	
Secuencia normal	
Secuencias alternativas	
Requisitos relacionados	

Tabla 5: Tabla tipo para los casos de uso

1. Cada caso de uso tendrá un identificador con el formato CU-XX, en que XX será un número de dos cifras que lo identifique unívocamente.
2. Título será un pequeño texto que describa a grandes rasgos el caso de uso, coincidirá con el expresado en la **ILUSTRACIÓN 5**.
3. La descripción será la especificación textual e intuitiva del caso de uso.
4. Las precondiciones de un caso de uso son las características de la situación que debe darse para que el proceso del caso de uso se lleve a cabo.
5. Las postcondiciones son las características de la situación que se genera después de completar el caso de uso.
6. Secuencia normal describe los procesos a llevar a cabo en el sistema para completar el caso de uso.
7. Secuencia alternativa describe posibles procesos diferentes del caso normal.
8. Los requisitos relacionados son aquéllos que definen procesos o elementos que aparecen en este caso de uso.

5.3.1 Especificación de casos de uso

CU-01	
Título	Inicialización de la biblioteca
Descripción	Para poder utilizar las funciones de la biblioteca se debe inicializar un objeto de la clase dcpl::inicializador. Que debe recibir como argumentos argc y argv.
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa.
Postcondiciones	1. La biblioteca estará lista para utilizar el resto de funciones. 2. Se habrá recopilado la información del contexto. 3. Se habrá creado el objeto dcpl::cout
Secuencia normal	1. Crear un método main en el programa que reciba como argumentos: `int argc` y `char** argv`. 2. Invocar al constructor de la clase dcpl::inicializador con dichos argumentos.
Secuencias alternativas	1. Crear un método main en el programa que reciba como argumentos: `int argc` y `char** argv`. 2. Invocar al constructor de la clase dcpl::inicializador con argumentos diferentes. 3. El comportamiento sería **no definido.**
Requisitos relacionados	REQ-SI-04, REQ-SI-03, REQ-SI-23

Caso de uso 01: Inicialización de la biblioteca

CU-02	
Título	Instanciar vector distribuido con reparto bloque
Descripción	El usuario instanciará un dcpl::DistributedVector
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador.
Postcondiciones	1. El vector instanciado estará en un estado válido. 2. El modo de reparto del vector estará guardado en el mismo. 3. El tamaño del vector será 0 elementos.
Secuencia normal	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de bloque. 2. Pasar los argumentos requeridos: tipo de reparto.
Secuencias alternativas	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de bloque. 2. Pasar los argumentos requeridos: tipo de reparto incorrectamente, pasando un tipo de reparto que no exista. 3. El comportamiento de la biblioteca será **no definido**.
Requisitos relacionados	REQ-SI-02

Caso de uso 02: Instanciar vector distribuido con reparto bloque

CU-03	
Título	Instanciar vector distribuido con reparto Round Robin.
Descripción	El usuario instanciará un dcpl::DistributedVector
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador.
Postcondiciones	1. El vector instanciado estará en un estado válido. 2. El modo de reparto del vector estará guardado en el mismo. 3. El tamaño del vector será 0 elementos.
Secuencia normal	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de ROUND ROBIN. 2. Pasar los argumentos requeridos: tipo de reparto: tamaño de rodaja del reparto.
Secuencias alternativas	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de round Robin. 2. Pasar argumentos no válidos, tanto un tipo de reparto que no sea round Robin como un tamaño de rodaja inválido. Un tamaño de rodaja inválido es todo número menor que 0 o no entero. 3. El comportamiento será **no definido.**
Requisitos relacionados	REQ-SI-02

Caso de uso 03: Instanciar vector distribuido con reparto round Robin.

CU-04	
Título	Instanciar vector distribuido con reparto *ad-hoc*.
Descripción	El usuario instanciará un dcpl::DistributedVector
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador.
Postcondiciones	1. El vector instanciado estará en un estado válido. 2. El modo de reparto del vector estará guardado en el mismo. 3. El tamaño del vector será 0 elementos. 4. La longitud de los bloques será almacenada en el vector.
Secuencia normal	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de *ad-hoc*. 2. Pasar los argumentos requeridos: tipo de reparto y vector de longitudes de los bloques.
Secuencias alternativas	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de *ad-hoc*. 2. Pasar los argumentos requeridos: tipo de reparto y vector de longitudes de los bloques no válidas. (ver: **REQ-SI-02**)
Requisitos relacionados	REQ-SI-02

Caso de uso 04: Instanciar vector distribuido con reparto ad-hoc.

CU-05	
Título	Instanciar vector distribuido con reparto *benchmark*.
Descripción	El usuario instanciará un dcpl::DistributedVector
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador.
Postcondiciones	1. El vector instanciado estará en un estado válido. 2. El modo de reparto del vector estará guardado en el mismo. 3. El tamaño del vector será 0 elementos. 4. Se realizarán rutinas para que al finalizar el programa se incluyan en un fichero los tiempos de ejecución del mismo.
Secuencia normal	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de BENCHMARK. 2. Pasar los argumentos requeridos: tipo de reparto y vector de longitudes de los bloques.
Secuencias alternativas	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de ad-hoc. 2. Pasar los argumentos requeridos incorrectamente, un tipo de reparto que no sea *ad-hoc* o un vector de longitudes que no cumpla las condiciones.
Requisitos relacionados	REQ-SI-02, REQ-SI-05

Caso de uso 05: Instanciar vector distribuido con reparto benchmark.

CU-06	
Título	Instanciar vector distribuido con reparto optimizado.
Descripción	El usuario instanciará un dcpl::DistributedVector
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador. 4. Haber ejecutado previamente el modo *benchmark* en este conjunto de máquinas.
Postcondiciones	1. El vector instanciado estará en un estado válido. 2. El modo de reparto del vector estará guardado en el mismo. 3. El tamaño del vector será 0 elementos. 4. Se habrán guardado en el contexto los tiempos de ejecución desde el archivo
Secuencia normal	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de optimizado. 2. Pasar los argumentos requeridos: tipo de reparto.
Secuencias alternativas	1. Invocar al constructor de la clase dcpl::DistributedVector correspondiente al modo de reparto de ad-hoc. 2. Pasar los argumentos requeridos incorrectamente: un tipo de reparto que no sea optimizado o un vector de longitudes que no cumplan las condiciones.
Requisitos relacionados	REQ-SI-02, REQ-SI-05

Caso de uso 06: Instanciar vector distribuido con reparto optimizado.

CU-07	
Título	Llenado del vector desde archivo binario
Descripción	El usuario utilizará un objeto de la clase dcpl::ifstream para llenar el vector con los elementos del archivo binario especificado en la instanciación del ifstream.
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador. 4. Haber instanciado un vector correctamente.
Postcondiciones	1. El vector pasado como argumento contendrá, con el modo de reparto seleccionado en su instanciación, los elementos leídos.
Secuencia normal	1. Se instancia un objeto de la clase dcpl::ifstream con la ruta al archivo a leer como argumento, ej.: `dcpl::ifstream ifs =dcpl::ifstream("./DATA")` 2. Se llama a dcpl::ifstream::read con el vector que se desea leer y el número de elementos que se desean leer como argumentos: (`ifs.read(v, elementos)`) 3. Si `elementos` es menor o igual que el número de datos contenido en el fichero `DATA`, se leerán ese número de elementos, si no, se leerán los elementos especificados por el argumento.
Secuencias alternativas	1. Si el segundo argumento de la llamada a dcpl::ifstream::read es negativo o 0, el funcionamiento será **no definido**.
Requisitos relacionados	REQ-SI-10, REQ-SI-11

Caso de uso 07: Llenado del vector desde archivo binario

CU-08	
Título	Escritura del contenido del vector en archivo binario.
Descripción	El usuario escribirá N elementos del vector distribuido en un archivo binario de su elección.
Precondiciones	1. Haber incluido la biblioteca 2. Tener un método *main* en el programa. 3. Haber instanciado un objeto de la clase dcpl::inicializador. 4. Haber instanciado un vector correctamente.
Postcondiciones	El archivo binario elegido contendrá los N primeros elementos del vector. Siendo N parámetro del usuario. Si N es mayor que la longitud del vector, entonces sólo se escribirán los elementos del vector.
Secuencia normal	1. Se instancia un objeto de la clase dcpl::ifstream con la ruta al archivo a escribir como argumento, ej.: `dcpl::ifstream ifs =dcpl::ifstream("./DATA")` 2. Se llama a dcpl::ifstream::write con el vector y el número de elementos que se desean escribir como argumentos: (`ifs.write(v, elementos)`) 3. Si `elementos` es menor o igual que el número de datos contenido en el vector v, se escribirán ese número de elementos, si no, se escribirán todos los elementos que el vector contiene.
Secuencias alternativas	Si el argumento `elementos` es 0 o negativo el comportamiento es **no definido**.
Requisitos relacionados	REQ-SI-10, REQ-SI-11

Caso de uso 08: Escritura del contenido del vector en archivo binario.

CU-09	
Título	Uso del iterador
Descripción	El usuario conseguirá un iterador de inicio y de fin de un dcpl::DistributedVector.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1
Postcondiciones	1. Se tendrán iteradores válidos de tipo *forward*. Apuntando uno al primer elemento del vector y el otro al siguiente del último elemento.
Secuencia normal	1. Se invoca al método `dcpl::DistributedVector::begin()` y se guarda el resultado en una variable. 2. Se invoca al método `dcpl::DistributedVector::end()` y se guarda en otra variable.
Secuencias alternativas	1. Si en vez de un iterador de inicio se requiere un iterador apuntando cualquier posición del vector, se puede utilizar el método `std::advance(iterador);` para cambiar la posición del mismo.
Requisitos relacionados	—

Caso de uso 09: Uso del iterador

CU-10	
Título	Uso de reduce
Descripción	El usuario utilizará el método reduce para calcular una operación de reducción sobre un vector distribuido o parte del mismo.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	5. Se obtendrá el resultado de la operación.
Secuencia normal	1. Obtener iterador de inicio. 2. Obtener iterador de fin 3. Elegir un objeto invocable para aplicarlo como operador binario (puede ser un objeto con operador paréntesis sobrecargado, una función lambda o un puntero a función) 4. Elegir un parámetro inicial (*init*) 5. Invocar al método reduce con los iteradores obtenidos, el parámetro inicial el operador.
Secuencias alternativas	1. Obtener un iterador de inicio 2. Avanzarlo con `std::advance` o con los operadores de incremento del mismo. 3. Obtener otro iterador de inicio y avanzarlo de igual modo para que apunte a una posición posterior a la del primero. Si el iterador *last* es anterior a *first*, la función nunca retornará. 4. Pasos del 3 al 5 de la secuencia normal.
Requisitos relacionados	REQ-SI-14, REQ-SI-15, REQ-SI-16

Caso de uso 10: Uso de reduce

CU-11	
Título	Uso de *transform*
Descripción	El usuario de la biblioteca utiliza el método dcpl::transform sobre un vector distribuido o una parte.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	Si los argumentos de la función son: first, last, result y op. (**REQ-SI-18**) Los elementos en el intervalo [first, last) serán escritos en las posiciones a las que apunte result y siguientes después de aplicarles op.
Secuencia normal	1. Obtener iterador de inicio. 2. Obtener iterador de fin. 3. Obtener iterador de inicio de otro vector o del mismo. 4. Avanzar los iteradores con `std::advance` o con los operadores oportunos. 5. Elegir un objeto invocable que haga las veces de operador unario. 6. Invocar la función `dcpl::transform (iterador1, iterador2, iterador_resultado, operador)`
Secuencias alternativas	Si después de los pasos del 1 al 4, ambos incluidos, `iterador1` es posterior a `iterador2`, el método no retornará. Si se da que el rango entre `iterador1` e `iterador2` es mayor que el rango entre `iterador_resultado` y el final del vector al que referencia, el comportamiento no está definido.
Requisitos relacionados	REQ-SI-18

Caso de uso 11: Uso de transform

CU-12	
Título	Uso de transform simple
Descripción	En este caso el usuario invoca la función con la finalidad de aplicar la transformación a todo un vector y guardar el resultado en dicho vector. Esto invocará una ejecución optimizada para este caso.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	Las mismas que para el CU-11
Secuencia normal	1. Conseguir el iterador de inicio de un vector (a= `v.begin()`) 2. Conseguir el iterador de fin de un vector (b = `v.end()`) 3. Elegir un objeto invocable que haga las veces de operador unario. (`operador`) 4. Invocar la función `dcpl::transform(a, b, a, operador)`
Secuencias alternativas	—
Requisitos relacionados	REQ-SI-18

Caso de uso 12: Uso de transform simple

CU-13	
Título	Uso de set
Descripción	El método set tiene la función de escribir un cambio en una posición del vector. El usuario lo utilizará en lugar del operador de acceso.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	1. La posición indicada contendrá el valor pasado como argumento.
Secuencia normal	1. Se elige una posición del vector. 2. Se elige un valor que introducir en ella. 3. Se invoca al método set de este modo: `set(posición, valor);`
Secuencias alternativas	Si la posición no está en el rango de posiciones válidas del vector (desde 0 a v.size() -1), el comportamiento es no definido.
Requisitos relacionados	REQ-SI-08

Caso de uso 13: Uso de set

CU-14	
Título	Uso de get
Descripción	El método get devuelve el valor de la posición indicada en el nodo indicado. El usuario lo utilizará para leer un valor del vector.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	1. El usuario dispondrá del valor de la posición indicada sólo en el nodo indicado. 2. El resto de nodos tendrá un valor de 0.
Secuencia normal	1. Elegir un nodo. 2. Elegir el vector sobre el que realizar la operación. (por ej.: v) 3. Elegir una posición dentro del rango del vector. 4. Invocar a get `a =v.get(posición, nodo);`
Secuencias alternativas	Si la posición está fuera de rango o el nodo no existe, el comportamiento es no definido.
Requisitos relacionados	REQ-SI-08

Caso de uso 14: Uso de get

CU-15	
Título	Utilización del operador[]
Descripción	El usuario utilizará el operador de acceso para realizar operaciones de entrada/salida sobre el vector asegurando la coherencia.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento.
Postcondiciones	1. El operador retornará una referencia al elemento en la posición indicada en el nodo que la tenga según el modo de reparto. 2. En otro caso devolverá una referencia a un miembro de clase del vector.
Secuencia normal	1. Elegir un vector sobre el que aplicar la operación. 2. Elegir una posición. 3. Elegir un valor que introducir en la posición. 4. Invocar a set: `v.set(posición, valor);`
Secuencias alternativas	Si la posición no existe, el comportamiento es no definido.
Requisitos relacionados	REQ-SI-08, REQ-SI-07

Caso de uso 15: Utilización del operador[]

CU-16	
Título	Referenciar un iterador
Descripción	El usuario utilizará un iterador para acceder a un elemento del vector.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector. 4. Haberlo llenado con al menos 1 elemento. 5. Haber obtenido un iterador del vector.
Postcondiciones	1. El usuario dispondrá de una referencia al elemento en memoria en el nodo que lo aloje según el tipo de reparto. 2. En el resto de nodos se devolverá a una referencia a un miembro de la instancia vector.
Secuencia normal	1. El usuario referencia el iterador con el operador asterisco: `*it`
Secuencias alternativas	Si el iterador no apuntara a una posición válida, el comportamiento sería no definido. Es el caso de iterador de final (v.end())
Requisitos relacionados	REQ-SI-07

Caso de uso 16: Referenciar un iterador

CU-17	
Título	Comparar iteradores
Descripción	El usuario hará uso de los operadores de comparación entre iteradores (igualdad, ==; y desigualdad, !=).
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector o varios. 4. Haber obtenido al menos dos iteradores de un vector, o del mismo.
Postcondiciones	1. El operador de igualdad devolverá un valor lógico que será cierto si solo si los dos iteradores apuntan a la misma posición del mismo vector. 2. El operador de desigualdad tendrá el comportamiento contrario.
Secuencia normal	1. Utilizar una operación de comparación entre iteradores. 2. Guardar su valor.
Secuencias alternativas	—
Requisitos relacionados	REQ-US-09

Caso de uso 17: Comparar iteradores

CU-18	
Título	Preincrementar
Descripción	El usuario utilizará un operador de preincremento con un iterador para que apunte a la posición siguiente, recuperará el valor del mismo después de dicha operación.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector como mínimo. 4. Haber obtenido al menos un iterador de dicho vector.
Postcondiciones	1. El iterador apuntará a la posición siguiente a la que apuntaba 2. La sentencia tendrá el valor del iterador incrementado.
Secuencia normal	1. Se aplica el operador a un iterador y se guarda el valor. `new_it = ++it;` 2. Si `it` apuntaba a la posición 0, ahora tanto `it` como `new_it` apuntan a la posición 1.
Secuencias alternativas	—
Requisitos relacionados	REQ-US-09

Caso de uso 18: Preincrementar

CU-19	
Título	Postincrementar
Descripción	El usuario utilizará un operador de postincremento con un iterador para que apunte a la posición siguiente, recuperará el valor del mismo antes de dicha operación.
Precondiciones	1. Tener un método *main* en el programa. 2. Haber instanciado un dcpl::inicializador. 3. Haber instanciado un vector como mínimo. 4. Haber obtenido al menos un iterador de dicho vector.
Postcondiciones	1. El iterador apuntará a la posición siguiente a la que apuntaba 2. La sentencia tendrá el valor del iterador antes del incremento.
Secuencia normal	1. Se aplica el operador a un iterador y se guarda el valor. `new_it = it++;` 2. Si `it` apuntaba a la posición 0, ahora `it` apunta a la posición 1 pero `new_it` apunta a la posición 0.
Secuencias alternativas	—
Requisitos relacionados	REQ-US-09

Caso de uso 19: Postincrementar

5.4 Diseño lógico

A continuación, se van a describir todos los componentes *software* que forman la biblioteca y sus atributos, operaciones y relaciones. Se va a utilizar una notación de diagrama de clases. En la Ilustración 6 se pueden ver todas las clases y las relaciones entre ellas. Además, se ha incluido el conjunto de operaciones que poseen:

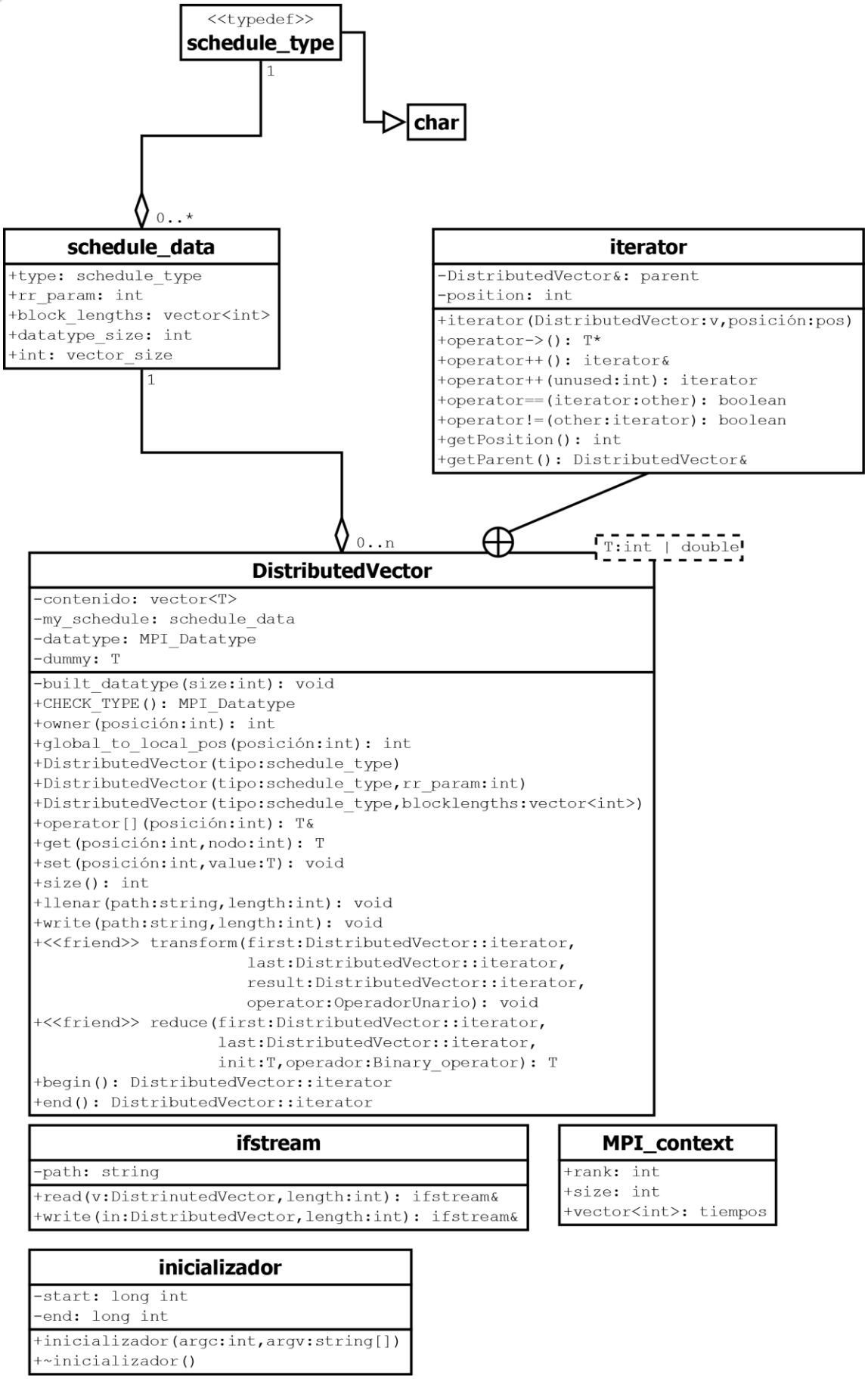


Ilustración 6: Diagrama de clases de la biblioteca

Siendo este el diseño de clases, la biblioteca cuenta con las siguientes variables globales:

1. PER_INFO_PATH: Es una constante que define la ubicación en la que se tienen que guardar los ficheros de registro de rendimiento **[REQ-SI-04]**.
2. My_context: es una variable de tipo MPI_context que se utiliza como contexto de MPI de la biblioteca. **[REQ-SI-03]** Es la encargada de almacenar el identificador de cada proceso y el dato de cuántos procesos están ejecutando el sistema.
3. Benchmark_flag: Es una variable lógica que indica si cuando se termine el programa se deben escribir los datos de rendimiento **[REQ-SI-05]**
4. Cout: Es una variable de tipo std::ostream, es un stream de salida de C++ que, si nos encontramos en el proceso con identificador igual a 0, será una copia de std::cout, la salida estándar de C++, si no, será un ostream que no haga nada cuando se use para imprimir. **[REQ-SI-23]** Es una variable que permitirá al usuario elegir que las impresiones por la salida estándar se produzcan sólo por el nodo número 0.

Después de tener una visión general de las entidades de la biblioteca, se explicará la función de cada una y se describirá la semántica de sus operaciones. La tabla utilizada para describir una operación de clase será la siguiente:

ID	**<Clase>-<XX>**	
Nombre		
Semántica		
Argumentos	Nombre	Tipo
Algoritmo		

Tabla 6: Tabla tipo para las operaciones

Los campos tienen el siguiente contenido:

1. ID: será un identificador único de la operación dentro del sistema, se formará como una abreviatura de la clase y un número de dos cifras. Las abreviaturas serán, a su vez:

Clase	**Abreviatura**
Iterador	IT
DistributedVector	DV
Ifstream	FS
Inicializador	IN

Tabla 7: Abreviaturas de las clases que componen el sistema

2. Nombre: nombre de la operación tal como aparece en la **ILUSTRACIÓN 6**.
3. Semántica: Explicación concisa de la operación que se realiza en lenguaje natural.
4. Argumentos: Especificación del tipo y nombre de los argumentos recibidos que se utilizarán en la explicación en profundidad del algoritmo.
5. Algoritmo: Explicación en forma de pseudocódigo de la operación realizada.

5.4.1 MPI Context

Esta clase no es más que una estructura de datos que contiene el rango del proceso, el número de procesos ejecutando el programa y un vector que, en caso de ser necesario (si un vector se instancia en modo optimizado), almacenará los tiempos de ejecución leídos de un fichero **(REQ-SI-04)**.

5.4.2 Inicializador

Esta clase es la encargada de realizar los métodos que permiten que MPI funcione. En un programa MPI hay que realizar una llamada a MPI_Init al inicio y una a MPI_Finalize al final. Para evitar que el usuario de la biblioteca deba realizar esas llamadas, se le pide sólo instanciar un objeto de este tipo que en su constructor recibe los argumentos de la función *main* del programa y llama con ellos a MPI_Init. **(REQ-SI-02)**. Cuando el programa termine y se salga de la función *main*, el destructor, entre otras cosas, llamará a MPI_Finalize. La principal función de esta clase, además de la ejecución de las funciones anteriormente citadas es la de medir los tiempos de ejecución del programa. Sus operaciones son:

ID	**IN-01**	
Nombre	Inicializador	
Semántica	El constructor recibe como argumentos los mismos que la función *main* del programa, y llama a MPI_Init con ellas. Además, llama a rutinas de MPI para conseguir el identificador de cada proceso y el número de procesos involucrados en la ejecución. Registra el tiempo de inicio del programa y lo guarda en su miembro *start*. Finalmente, detecta si éste es el proceso 0 para que la variable cout del *namespace* tenga el valor correcto. (REQ-SI-23: Impresión sólo por parte del nodo o proceso 0)	
Argumentos	Nombre	Tipo
	argc	int
	argv	char **
Algoritmo	this.start = medirTiempo() MPI_Init(argc, argv) my_context.rank = MPI_Rank() mycontext.size=MPI_Size() SI(my_context.rank = 0) dcpl::cout = std::cout	

Operación 1: IN-01

ID	IN-02
Nombre	~inicializador
Semántica	Si la variable benchmark_flag es cierta, se registran los tiempos de ejecución de todos ellos en un fichero, de lo contrario no se hace nada. Después de lo anterior, en cualquier caso, se llama a MPI_Finalize. Para mantener la coherencia si el sistema de ficheros no es compartido, todos los procesos reciben los tiempos de los demás, lo que se realiza en una operación de *broadcast*.
Argumentos	—

Algoritmo	SI (benchmark_flag = **false**)
	MPI_Finalize();
	return;
	this.end = medirTiempo();
	resúmenes = Resumir_nombres_nodos();
	int tiempos[my_context.size];
	tiempos[my_context.rank] = this.end-this.start;
	Broadcast (tiempos[my_context.rank])
	tiempos = Recibir_tiempos;
	archivo = Abrir_archivo (PER_INFO_PATH)
	archivo.escribir(tiempos, resúmenes);

Operación 2: IN-02

5.4.3 Clase *Schedule_data*

Esta clase es la encargada de almacenar el modo de reparto de cada vector distribuido (REQ-SI-01). Es una estructura de datos y, por tanto, tiene todos sus miembros públicos. Dichos miembros son:

1. *Type*: Indica el tipo de reparto, para que la escritura sea más clara, se ha definido un nuevo tipo de datos llamada *schedule_type*, que es un alias del tipo char. Los valores válidos para este miembro serán:

Significado	Valor
BLOQUE	1
ROUND ROBIN	2
BENCHMARK	3
AH_HOC	4

Tabla 8: Valores válidos y su significado del tipo de reparto

2. Parámetro de *Round Robin*: en caso de que el reparto cíclico (también llamado de *Round Robin*) sea elegido para un vector, el tamaño de la rodaja se almacenaría en este campo.
3. Block_lengths: Si el tipo de reparto de un vector es *ad-hoc* u optimizado, aquí se almacenarán las longitudes de los bloques de datos que cada proceso alojará.

4. Datatype Size: Aquí se guardará el tamaño del tipo de datos de MPI que se utilizará para leer el archivo. Se le dará valor en el momento de llenar el vector con el contenido de un archivo binario.

La clase no contiene operaciones y todos sus miembros son públicos, esto es debido a que es un *struct*, que es el equivalente en C y C++ a una clase de esas características.

5.4.4 Clase *iterator*

Esta clase es la encargada de implementar el iterador de acceso al vector distribuido. Sus miembros son:

1. *Parent*: Es una referencia al vector al que apunta.
2. Position: La posición de dicho vector a la que apunta.

El mayor contenido de la clase reside en sus operaciones, que son mayormente operadores, esto es porque son éstos operadores los que implementarán la funcionalidad de la clase para que se ajuste a los requisitos del sistema (REQ-SI-14), dichas operaciones son:

ID	**IT-01**	
Nombre	iterator	
Semántica	Instancia un iterador apuntando a un vector y a una posición del mismo.	
Argumentos	Nombre	Tipo
	v	DistributedVector
	pos	Int
Algoritmo	this.parent = v; this.position = pos;	

Operación 3: IT-01

ID	**IT-02**
Nombre	Operator->
Semántica	Devuelve un puntero al contenido de la posición que referencia el iterador sobre el que se aplica. Tal y como está diseña el sistema, si el nodo sobre el que se aplica aloja la posición en memoria, devuelve el puntero a esa posición, si no, pide el dato al nodo que lo tenga y devuelve una referencia **[2.1]** al miembro *dummy* de la clase DistributedVector.
Argumentos	—
Algoritmo	return dirección de: parent[position];

Operación 4: IT-02

ID	IT-03	
Nombre	Operator++	
Semántica	Este operador debe incrementar en una unidad la posición referenciada por el iterador. No se hace comprobación de rango alguna, así que con este operador se podrían lograr iteradores fuera del rango del vector. El resultado del operador será una referencia al iterador con el valor **después** del incremento.	
Argumentos	—	
Algoritmo	this.position = this.position + 1; return this;	

Operación 5: IT-03

ID	IT-04	
Nombre	Operator++	
Semántica	Se comporta exactamente igual que el operador de preincremento, pero devuelve una copia del iterador sobre el que se aplica, con el valor **antes** del incremento.	
Argumentos	Nombre	Tipo
	unused[1]	int
Algoritmo	copia = this; ++this; return copia;	

Operación 6: IT-04

ID	IT-05	
Nombre	Operator==	
Semántica	Comprueba que dos iteradores son iguales. La implementación comprueba que la posición a la que referencia (*position*) es igual y, además, que la dirección del vector en memoria es la misma. Esta implementación hace que dos iteradores referenciando a la misma posición de vectores con el mismo contenido sean diferentes.	
Argumentos	Nombre	Tipo
	iterator	other
Algoritmo	pos_cierta = this.position == other.position direcc_cierta = dirección(this.parent) ==dirección(other.parent) return poscierta AND direcc_cierta	

Operación 7: IT-05

[1] Para diferenciar el operador preincremento del postincremento, C++ define este último con un argumento de tipo int que no tiene utilidad. **[20]**

ID	IT-06	
Nombre	Operator!=	
Semántica	Devuelve el resultado contrario al operador IT-04.	
Argumentos	Nombre	Tipo
	iterator	other
Algoritmo	return NOT(this==other)	

Operación 8: IT-06

ID	IT-07
Nombre	getPosition
Semántica	Devuelve el número de la posición a la que referencia el iterador. Usado por los algoritmos. Es público para que puedan acceder a ellos, aunque no se prevé que el usuario lo utilice, no supone riesgo para la integridad de los datos del iterador puesto que no lo modifica.
Argumentos	—
Algoritmo	return this.position;

Operación 9: IT-07

ID	IT-08
Nombre	getParent
Semántica	Devuelve la referencia al vector al que pertenece el elemento al que apunta el iterador.
Argumentos	—
Algoritmo	return this.parent;

Operación 10: IT-08

5.4.5 DistributedVector

Es la clase principal de la biblioteca, que permite almacenar en manera distribuida según los tipos de reparto disponible un conjunto de datos leídos de un archivo binario. Esta clase **[ILUSTRACIÓN 6]** tiene los siguientes miembros:

- Contenido: Es un vector estándar de C++ (std::vector) que contendrá en memoria los datos que este proceso aloje en memoria.
- My_schedule: Es un miembro del tipo *schedule_data*, este miembro almacenará toda la información correspondiente al reparto de los elementos de este vector en los diferentes procesos.
- Datatype: Es un miembro del tipo MPI_Datatype, este miembro almacenará el tipo de datos de MPI utilizado para leer el archivo.
- Dummy: Este miembro se utiliza para el correcto funcionamiento de los iteradores y del operador de acceso al vector (operador []). Cuando en un proceso que no aloja cierta posición del vector se solicita, ya sea mediante iterador o mediante el operador, dicha posición, el proceso que la tiene manda el valor a todos los demás y éstos copian ese valor a este miembro. Así, una vez copiado el valor, pueden devolver tanto un puntero como una referencia al mismo sin problema. Por ejemplo: en un reparto de bloque entre 2 procesos (proceso 0 y proceso 1) con un vector de 100 elementos, si se solicita acceso a la posición 75, el proceso 1 mandará el valor al proceso 0 y éste copiará dicho valor en Dummy. Después, devolverá una referencia a dicho miembro de la clase.

Las operaciones que se pueden realizar sobre la clase son la siguientes:

ID	DV-01	
Nombre	build_datatype	
Semántica	Recibe la cantidad de datos que se desean leer. Es una función privada porque su semántica es la de generar el tipo de datos derivado que se utilizará para leer el archivo y que será almacenado en el miembro *datatype*. (REQ-SI-06)	
Argumentos	Nombre	Tipo
	size	int
Algoritmo	Según el tipo de reparto SI (reparto == BLOQUE) Elementos totales = size/tamaño(int); Cada proceso recibirá: elementos totales/número de procesos (redondeado a la baja) Si soy el último proceso recibo: los mismo que los demás más los que quedaran sin repartir por redondeo. this.datatype = Crear tipo de dato MPI de bloque con la longitud correspondiente. SI (reparto == ROUND ROBIN) Elementos totales = size/tamaño(int); Primera rodaja = my_context.rank*tamaño_rodaja MIENTRAS (haya elementos disponibles) Comprobar si se puede asignar una rodaja entera. Asignarla entera o todos los elementos que se puedan de la misma hasta completar el vector Ir a la siguiente rodaja. Crear tipo de datos discontinuo de MPI con los datos de rodajas asignadas. SI (reparto == OPTIMIZADO) Repartir inversamente los elementos del vector entre los procesos según su tiempo de ejecución. Convertir el vector en tipo AD-HOC Si (reparto == AD-HOC) Calcular las posiciones de inicio y fin del bloque debo leer. Crear tipo de dato continuo de MPI en esas posiciones. this.datatype = tipo de dato creado.	

Operación 11: DV-01

ID	DV-02
Nombre	CHECK_TYPE
Semántica	Si el vector recibió como argumento un tipo entero, esta función devolverá el tipo de datos de MPI correspondiente a los enteros (MPI_INT), si es un vector de double, devolverá MPI_DOUBLE. (REQ-SI-09)
Argumentos	—
Algoritmo	SI (tipo plantilla == int) Return MPI_INT SI (tipo plantilla == double) Return MPI_DOUBLE

Operación 12: DV-02

ID	DV-03	
Nombre	Owner	
Semántica	Es un método usado para conocer el proceso que aloja la posición que se pase como argumento	
Argumentos	Nombre	Tipo
	pos	int

Algoritmo	SI (reparto == BLOQUE)
	elementosPorProceso =
	tamañoVector /procesos;
	return pos -
	my_context.rank*elementosPorVector
	SI (reparto == ROBIN)
	rodaja = pos/tamaño rodaja
	return resto (rodaja, my_context.size)
	SI (reparto == AD_HOC)
	acumulado = 0;
	for(ii de 0 a my_schedule.block_lengths.size())
	SI(acumulado <= pos
	<acumulado+block_lengths[ii])
	return ii
	acumulado =
	acumulado+block_lengths[ii]	

Operación 13: DV-03

ID	DV-04	
Nombre	Global_to_local_pos	
Semántica	Es un método que recibe la posición global del vector y devuelve aquélla que ese elemento ocupa en el miembro **contenido**. Sólo funciona si se ha comprobado que el proceso llamante es el que aloja la posición.	
Argumentos	Nombre	Tipo
	Pos	int
Algoritmo	SI (reparto = BLOQUE) elemPerProcess = myschedule_vector_size/my_context.size return pos – my_context.rank*elemPerProcess SI (reparto = ROBIN) Tamaño de un reparto entero = tamañoRodaja*my_context.size Repartos hechos = pos/reparto_size Posición en la rodaja = pos – (repartos hechos*tamaño de un reparto) –(my_context.rank*ramañoRodaja) return repatos_enteros*tamañoRodaja+Posición en la rodaja SI (reparto = AH_HOC) SI (my_context.rank ==0) acumulado = 0; SI NO acumulado = Sumar todas las posiciones de my_schedule.block_lengths hasta: my_context.rank–1.	

Operación 14: DV-04

ID	DV-05	
Nombre	DistributedVector	
Semántica	Constructor para BLOQUE, BENCHMARK y OPTIMIZED.	
Argumentos	Nombre	Tipo
	tipo	schedule_type
Algoritmo	my_schedule.type = tipo SI (tipo == BENCHMARK) My_schedule.type = BLOCK Benchmark_flag = true; SI (tipo == OPTIMIZED) Comprobar si en PER_INFO_PATH existe. SI (No existe) Imprimir error Lanzar vector en modo bloque Se extraen del archivo tiempos y resúmenes de los nombres de los nodos. Se comprueba si estos nodos son una permutación de la ejecución anterior (mismos nodos en otro orden) Se emparejan los tiempos del archivo con los nodos actuales. Se guardan los tiempos en my_context.tiempos	

Operación 15: DV-05

ID	DV-06	
Nombre	DistributedVector	
Semántica	Constructor para Round Robin	
Argumentos	Nombre	Tipo
	tipo	schedule_type
	Rodaja	int
Algoritmo	this.my_schedule.type = tipo this.my_schedule.rr_param = rr_param	

Operación 16: DV-06

ID	DV-07	
Nombre	DistributedVector	
Semántica	Constructor para *ad-hoc*	
Argumentos	Nombre	Tipo
	tipo	schedule_type
	blocklengths	std::vector<int>
Algoritmo	this.my_schedule.type = tipo SI (tipo != AD_HOC OR blocklenghts.size != my_context.size) Lanzar excepción this.my_schedule.block_lengths = blocklengths	

Operación 17: DV-07

ID	DV-08	
Nombre	Operator[]	
Semántica	Este operador de comporta como el de la clase vector estándar de C++. Devuelve una referencia al elemento referenciado, permitiendo usar este operador para lectura o para escritura.	
Argumentos	Nombre	Tipo
	Pos	int
Algoritmo	local_pos = global_to_local_pos(pos) SI (my_context.rank == owner(pos)) Enviar en broadcast el dato a los demás procesos. return this.contenido[local_pos] SI NO Recibir el dato y alojarlo en this.dummy return this.dummy	

Operación 18: DV-08

ID	DV-09	
Nombre	Get	
Semántica	Esta función devuelve el valor del elemento que se indica, pero sólo en el proceso indicado como argumento, en el resto se devuelve 0.	
Argumentos	Nombre	Tipo
	posición	int
	nodo	int
Algoritmo	Soy receptor si: nodo == my_context.rank; Soy emisor si: owner(pos) == my_context.rank SI (soy emisor AND soy receptor) return this.contenido(global_to_local_pos(pos)) SI (sólo soy emisor) Envío el dato sólo al proceso receptor; SI (sólo soy receptor) Recibo el dato return dato	

Operación 19:DV-09

ID	DV-10	
Nombre	Set	
Semántica	Esta función hace que el nodo que aloja la posición cambie su contenido en memoria sin intervención de otros procesos.	
Argumentos	Nombre	Tipo
	Pos	Int
	Value	T (tipo de la plantilla)
Algoritmo	SI (owner(pos) != my_context.rank) return; this.contenido[global_to_local_pos] = value;	

Operación 20: DV-10

ID	DV-11
Nombre	size
Semántica	Devuelve el tamaño del vector distribuido, es decir, la suma del número de elementos alojados en todos los procesos.
Argumentos	—
Algoritmo	Return this.my_schedule.vector_size

Operación 21: DV-11

ID	DV-12	
Nombre	llenar	
Semántica	Es un método público que sirve para llenar con N datos leídos del fichero pasado cuya ruta se pasa como argumento. Será llamado por los métodos de la clase dcpl::ifstream.	
Argumentos	Nombre	Tipo
	path	string
	length	int
Algoritmo	Abrir con MPI el fichero apuntado por path SI (archivo.size() >= length) build_datatype(length) SI NO build_datatype(archivo.size()) MPI_set_view(archivo, this.datatype); MPI_read(archivo, contenido, this.datatype_size);	

Operación 22: DV-12

ID	DV-13	
Nombre	Write	
Semántica	Se comporta como llenar, pero en vez de leer los datos, los escribe en el fichero binario.	
Argumentos	Nombre	Tipo
	path	string
	Length	int
Algoritmo	Abrir con MPI el fichero apuntado por path MPI_set_view(archivo, this.datatype); SI (reparto == ROBIN) SI(owner(length) == my_context.rank) length = global_to_local_pos(length) SI NO mientras (owner(length) != my_context.rank) EN OTRO CASO SI (owner (length) < my_context.rank) length = 0 SI NO SI (my_context.rank == owner(length)) length = global_to_local_pos(length) SI NO length = my_schedule.datatype_size MPI_Write(archivo, contenido, length)	

Operación 23: DV-13

ID	DV-14
Nombre	Begin
Semántica	Retorna un iterador que referencia al primer elemento del vector (elemento 0).
Argumentos	—
Algoritmo	it = iterator(0, this); return it;

Operación 24: DV-14

ID	DV-15
Nombre	End
Semántica	Retorna un iterador que referencia al elemento siguiente al último elemento del vector.
Argumentos	—
Algoritmo	it = iterator(this.size(), this); return it;

Operación 25: DV-15

5.4.6 Clase ifstream

Esta clase es la que implementa la interfaz de lectura para llenar el vector con elementos de un archivo binario. (REQ-SI-10, REQ-SI-11). La clase sólo almacenará en un miembro la ruta del archivo cuando se instancie. Finalmente, dispondrá de dos métodos:

ID	FS-01	
Nombre	read	
Semántica	Sirve para leer del archivo binario e introducir los elementos que el mismo contiene en el vector pasado como argumento.	
Argumentos	Nombre	Tipo
	in	DistributedVector
	length	int
Algoritmo	in.llenar(this.path, length); return this;	

Operación 26: IF-01

ID	FS-02	
Nombre	write	
Semántica	Análogamente a read, escribe en el fichero binario los N primeros elementos del vector.	
Argumentos	Nombre	Tipo
	in	DistributedVector
	length	int
Algoritmo	in.write(this.path, length); return this;	

Operación 27: IF-02

5.4.7 Algoritmos distribuidos

Además, las funciones estáticas (no miembros) de la clase DistributedVector, pero definidas como *friends*[2] de ésta son las encargadas de implementar los algoritmos *transfrom* y *reduce* (REQ-US-10). Esto se ha hecho así porque necesitan acceder a los miembros de la clase directamente.

Estas funciones se han hecho con la intención de adaptar los algoritmos homónimos de las mismas a un paradigma de memoria distribuida. Para hacer esto la base de la programación es hacer lo mismo que haría una función de la STL y, según sea necesario durante la ejecución, solventar los problemas derivados de que los datos no estén en el mismo nodo o proceso. El problema principal derivado de ello es que un proceso puede necesitar un dato que no tenga, por lo que deberá recibirlo del proceso correspondiente.

Estos algoritmos deben imitar la interfaz de la STL. Esta librería está basada en plantillas. Las plantillas son, como se describe en la Introducción a C++ una técnica de metaprogramación que permite la programación genérica en el lenguaje C++, generando plantillas de funciones y plantillas de clases. Esto nos permite aplicar un algoritmo a un elemento cuyo tipo no sabemos. Mientras que las sentencias formadas después de que, en tiempo de compilación, se sustituya la plantilla por el tipo de dato al que hemos aplicado el algoritmo sean correctas, el programa funcionaría correctamente.

La metaprogramación basada en plantillas permite definir símbolos en el código que el compilador sustituirá en tiempo de compilación por el valor que sea necesario [21]. Es el caso de todos los contenedores de la STL como los mapas asociativos o los propios vectores. El argumento pasado como plantilla se sustituye en el código y se crear esa clase para el tipo de dato requerido. Esto funciona de modo parecido a los tipos genéricos de otros lenguajes como Java [22].

Por ejemplo:

```
int doble(int a){
	return a*2;
}
```

Código 10: Ejemplo de algoritmo no genérico

Sería un algoritmo válido para todo tipo que admitiera el operador * (multiplicación) por el entero 2. Para no definir una versión para cada tipo, podemos establecer una plantilla de función del siguiente modo:

```
template <class number>
number doble(number a){
	return a*2;
}
```

Código 11: Algoritmo genérico con metaprogramación

Así, tanto si aplicamos la función a un número de coma flotante de doble precisión, un entero o cualquier tipo definido por el usuario que admita la expresión, la función

[2] Una función o clase *friend* de otra clase es aquélla que puede acceder a sus miembros privados. Se utiliza cuando el acoplamiento entre ambas es alto.[29]

sería válida. Esta técnica es la que se ha utilizado para la programación de los algoritmos distribuidos.

A continuación, se explicarán con pseudocódigo el funcionamiento de los algoritmos.

El algoritmo *transform* de la STL tendría un comportamiento similar a:

```
template <class iterator, class UnaryOperator>
iterator transform(iterator first, iterator last, iterator result,
UnaryOperator op){
      while(first != last){
            * result = op(*last);
            ++result;
            ++first;
      }
      return result;
}
```

Código 12: Comportamiento de transform de la STL

Como se puede ver, se limita a acceder al iterador apuntado por *first* y, después, aplicar la función *op*, el resultado se guarda en *result*; y esto se repite hasta que *first* es igual a *last*, es decir, para el rango [*first*, *last*).

Tal y como se ha diseñado el iterador de la clase DistributedVector, este algoritmo funcionaría, pero como el iterador utiliza a su vez el operador de acceso (DV-08) y éste ejecuta un *broadcast*, sería ineficiente usarlo. Por ello, lo que se ha hecho es comprobar en cada iteración si el proceso que necesita el dato (aquél que aloje la posición a la que apunte en ese momento *result*) lo tiene, y si no, se lo manda el que sí aloje el elemento al que apunte *first*. En el Código 13 se puede ver cómo funciona el algoritmo utilizado en la biblioteca:

```
template <class iterator, class UnaryOperator>
iterator transform(iterator first, iterator last, iterator result, UnaryOperator op){
      if(first == result and last == last.getParent().end() and  first == first.getParent().begin())
            std::transform(first.getParent().contenido.begin(), last.getParent().contenido.end(),
            first.getParent().contenido.begin(), op);
      while(first != last){
            if (alojo first and alojo result){
                  dato = Leer el elemento en first;
                  Elemento en result = op(a);
                  continue;
            }
            if(alojo first){
                  MPI_send: el elemento en first al proceso que aloje result
            }
            else{
                  dato = MPI_receive: Elemento alojado en first;
                  Elemento en result = op(dato);
            }
            ++result;
            ++first;
      }
      return result;
}
```

Código 13: Algoritmo de la función transform en la biblioteca

En el inicio del algoritmo se comprueba si la función se llama con los argumentos correspondientes a:

```
std::transform(v.begin(), v.end(), v.begin(), op);
```

Esto es así porque en esta situación, basta con que todos los procesos ejecuten la función sobre todo su vector local, así, a todos los elementos les sería aplicado el operador *op* en paralelo.

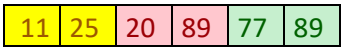
Si no nos encontramos en este caso, se hace algo muy parecido a lo que realiza la STL. Se accede al dato en el iterador *first*, se le aplica el operador *op* y se guarda en el elemento referenciado por *result*. Después tanto *first* y *result* se incrementan. Las únicas diferencias son: no se puede acceder mediante el operador de acceso al iterador y hay que comprobar qué proceso aloja el elemento apuntado por *first* y por *result*.

Para acceder a los elementos a los que referencian los iteradores no se usa su operador de acceso (operator*), sino que se accede directamente al miembro *contenido* de la clase DistributedVector. Esto nos permite evitar el *broadcast*. Un ejemplo sería:

```
Elemento en first =
first.getParent().contenido[global_to_local_pos(first.getPosition())]
```

Un ejemplo de funcionamiento del algoritmo sería:

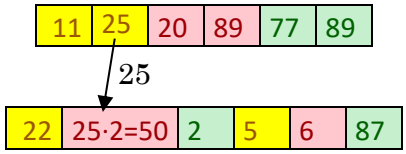
Un vector reparto en modo de bloque entre 3 procesos:



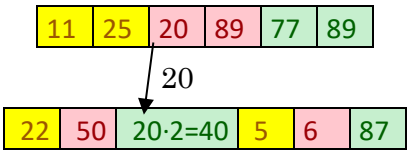
Un vector en modo round Robin con rodaja 1:

Si se desea multiplicar el primer vector por 2 y guardarlo en el segundo, las iteraciones serían:

1. El proceso amarillo aloja ambos datos: realiza la transformación localmente.
2. El proceso amarillo tiene el dato, será transformado y alojado en el proceso rojo, el proceso amarillo obtiene el valor de este modo:
    a) Recibe un iterador que apunta a la posición 1.
    b) Invoca a global_to_local_pos para saber en qué posición del vector en memoria está la posición 1. (en este caso, en la 1)
    c) Accede al valor en memoria a través del miembro contenido de la clase DistributedVector.
    d) Manda el dato al proceso rojo.



3. El proceso rojo opera con el dato recibido y lo guarda.
4. El proceso rojo tiene el dato, será transformado y alojado en el proceso verde.
    a) Recibe un iterador que apunta a la posición 2.
    b) Invoca a global_to_local_pos para saber en qué posición del vector en memoria está la posición 2 (en este caso, en la 0)
    c) Accede al valor en memoria a través del miembro contenido de la clase DistributedVector.
    d) Manda el dato al proceso verde.



Para el resto de posiciones se sigue el mismo proceso. Como se puede ver, al comprobar en cada paso qué proceso debe leer el elemento y cuál debe recibirlo, se realiza una comunicación punto a punto, por lo que no es necesario que los procesos ajenos a la comunicación intervengan.

El algoritmo *reduce* de la STL, por su parte, tendría un comportamiento similar a:

```
template <class InputIterator, class binaryOperator, class T >
T accumulate (InputIterator first, InputIterator last, T init, binaryOperator op){
  while (first!=last) {
    init=binary_op(init,*first);
    ++first;
  }
  return init;
}
```

Código 14: Comportamiento de reduce de la STL

Sin embargo, en este caso, MPI provee una función que nos permite aplicar una operación de reducción sobre un vector del mismo tamaño en todos los procesos. Dicha función se llama MPI Reduce. Teniendo una situación como la siguiente y aplicando la operación suma:

Nodo	V [0]	v [1]	v [2]	v [3]	v [4]	v [5]	v [6]
1	1	2	5	4	9	7	10
2	11	2	3	45	8	4	6
3	11	15	22	51	9	78	9
Resultado	23	19	30	100	26	89	25

Tabla 9: Ejemplo de utilización de MPI_Reduce

El resultado, a su vez, será compartido por todos los procesos al final de la operación.

Esto nos permite calcular localmente la reducción parcial para cada proceso y después utilizar MPI para sumar estos resultados, en la situación anterior:

Nodo	v [0]	v [1]	v [2]	v [3]	v [4]	v [5]	v [6]	Suma Parcial
1	1	2	5	4	9	7	10	38
2	11	2	3	45	8	4	6	79
3	11	15	22	51	9	78	9	195
							Resultado:	312

Tabla 10: Ejemplo de cálculo de sumas parciales en una reducción

Si la reducción parcial la calculamos en cada nodo, podemos usar MPI reduce para calcular la reducción total, aplicándolo a un elemento por nodo (vector de tamaño 1). El principal problema de esta técnica es que la operación MPI reduce sólo admite como operando un puntero a función. En C++ existen fundamentalmente 3 tipos de objeto ejecutable como una función (mediante paréntesis): un objeto con ese operador sobrecargado, una función lambda y un puntero a función. De este modo, hay que

convertir los dos primeros a un puntero a función para que pueda ejecutar. Luego los pasos a realizar para calcular una reducción con una operación determinada serían:

1. Cada proceso calcula su reducción aplicando a todos los elementos que posea y estén dentro del rango [*first, last*) la operación simbolizada por el operador ⋄.

   $$Suma\ parcial = v_1 \diamond v_2 \diamond v_3 \ldots \diamond v_n$$

   Si el vector es {1,2,3,4,5} y el operador fuera la suma, la operación sería: $1 + 2 + 3 + 4 + 5 = 15$.

2. Si un proceso no tiene elementos con los que operar dentro del rango, no entregará ningún resultado. Para esto:
   a. Los procesos con resultados válidos crean otro comunicador[3] aparte.
   b. Ejecutan MPI_reduce únicamente los procesos con resultados válidos.
   c. Mandan el dato a los procesos que no ejecutaron MPI_reduce para que tengan una versión actualizada del resultado.
3. Los procesos retornan el dato

[3] Un comunicador es la abstracción para un subconjunto de procesos que realizan las operaciones colectivas como un grupo aparte del resto **[23]**.

El Código 15 muestra este algoritmo en detalle:

```
1  template<class ForwardIt, class U, class Binary_op>
2  U reduce(ForwardIt first, ForwardIt last, U init, Binary_op binary_op){
3        static auto _op = binary_op;
4        class interOperator{
5              public:
6              static void addem(U* invec, U* inout, int* length, MPI_Datatype* dtype){
7
8                    for(int ii = 0; ii < *length; ++ii){
9                          inout[ii] = (_op(inout[ii], invec[ii]));
10                   }
11             }
12       };
13       auto Puntero a función = &(interOperator::addem)
14       if(my_context.rank == 0){
15             reducción parcial = init;
16             primer = false;
17       }
18       while(first != last){
19                   if(Alojo el elemento apuntado por first){
20                         if(primer){
21                               reducción parcial = elemento apuntado por first;
22                               ++first;
23                               continue;
24                         }
25                         reducción parcial = binary_op(reducción parcial,
26                         elemento apuntado por first);
27                   }
28                   ++first;
29       }
30       if(!primer){
31             Unirse a un nuevo comunicador de MPI
32             Operación = MPI_Create_Op(puntero a función)
33             Reducción_final = MPI Reduce (reducción parcial, operación)
34             Broadcast(Reducción_final);
35       }else{
36             recibir(Reducción_final)
37       }
38
39       return Reducción_final;
40 }
```

Código 15: Algoritmo de la función reduce en la biblioteca

Para crear una operación compatible con MPI Reduce se debe crear una función con un prototipo específico. Dicho prototipo es:

```
void addem(U* invec, U* inout, int* length, MPI_Datatype* dtype)
```

Siendo U el tipo de datos del vector al que se le aplique la reducción. Para lograr esto se debería crear una función según el operador elegido por el usuario. Pero no se pueden crear funciones dentro de funciones. La opción elegida fue, por tanto, crear una clase que tuviera un solo método estático. Esta clase (en líneas de la 4 a la 12 en el Código 15) puede utilizar el operador provisto por el usuario para conformar dicha función estática. Una vez hecho esto, se puede obtener el puntero a la función estática de la clase.

Esta función, por cómo se ha convertido en un puntero a función cualquier objeto invocable de C++, sería potencialmente usable con funciones lambda que capturen variables, por ejemplo, o con objetos función, pero esto no es así. Si se captura una variable, dicha variable es capturada en cada proceso con el valor que tenga en su memoria y, por tanto, en cada uno podría tener un valor diferente, lo que provocaría comportamientos no definidos. Lo mismo pasaría con los miembros de un objeto pasado como argumento.

En la línea 13 es donde se puede ver la conversión de la función estática a puntero a función. Por otro lado, en la 18 y siguientes se ve el bucle donde cada proceso calcula la reducción parcial que le corresponde. Todos los procesos comprueban si es la primera ejecución que realizan, para empezar con el dato inicial del rango, si es cualquier iteración subsiguiente, utilizan el dato de la iteración anterior para seguir calculando la reducción. Después del bucle, a partir de la línea 30, se crea un comunicador sólo entre los procesos que ejecutaron al menos una operación y, después, se ejecuta MPI reduce y se difunde el resultado a los demás procesos.

6. Pruebas de validación

En esta sección se van a proponer una serie de pruebas que permitan constatar que los requisitos de sistema han sido correctamente implementados en la aplicación. Toda prueba estará relacionada, al menos, con un requisito de sistema. Además, se comprobará el resultado de la prueba.

ID	PR-XX
Título	
Descripción	
Pasos	
Resultado esperado	
Prueba superada	Sí | No
Requisitos relacionados	

Tabla 11: Tabla tipo para especificación de una prueba

Los campos de la tabla tienen los siguientes significados:

1. ID: Identificador único de la prueba que tiene la forma PR-XX siendo XX un número de 2 cifras.
2. Un texto corto que referencie el contenido de la prueba
3. Descripción: Explicación en lenguaje natural del objetivo de la prueba.
4. Pasos: Acciones a realizar en el sistema para la realización de la prueba.
5. Resultado esperado: Reacción del sistema ante los pasos anteriores que se espera para superar la prueba.
6. Prueba superada: Indica si el sistema se comportó como se esperaba.
7. Requisitos relacionados: Requisitos de sistema probados en esta prueba.

6.1 Especificación de las pruebas

ID	PR-01
Título	Inicialización de la biblioteca
Descripción	En esta prueba se comprobará que la biblioteca es capaz de ejecutar las tareas de inicialización y finalización correctamente. Se comprobará por tanto que se inicia MPI con MPI_Init y que se finaliza con MPI_Finalize.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Retornar de la función *main*.
Resultado esperado	El programa indicará con mensajes de depuración la ejecución correcta de todas las funciones de MPI correspondientes.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-03, REQ-SI-19, REQ-SI-20, REQ-SI-21

PR-01: Inicialización de la biblioteca

ID	PR-02
Título	Instanciación de un vector con modo de reparto BLOQUE
Descripción	Se instanciará un vector de tipo bloque y se comprobará que la inicialización de sus estructuras de datos se hace correctamente.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un objeto DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto DistributedVector de doubles con tipo de reparto bloque. 4. Retornar de la función *main*.
Resultado esperado	El programa indicará con mensajes de depuración que el vector ha sido instanciado correctamente, es decir, que sus estructuras de datos han sido correctamente inicializadas.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-02, REQ-SI-07, REQ-SI-09

PR-2: Instanciación de un vector con modo de reparto BLOQUE

ID	PR-03
Título	Instanciación de un vector con modo de reparto Round Robin.
Descripción	Se comprobará que la instanciación de un vector de tipo Round Robin es correcta, con diferentes tamaños de rodaja (1, 10 y INT_MAX)
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto round Robin con rodajas tamaño 1, 10 e INT_MAX. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto round Robin con rodajas tamaño 1, 10 e INT_MAX. 4. Retornar de la función *main*.
Resultado esperado	El vector inicializará su modo de reparto con los datos correctos.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-02, REQ-SI-07

PR-3: Instanciación de un vector con modo de reparto Round Robin.

ID	PR-04
Título	Instanciación de un vector con modo de reparto BENCHMARK
Descripción	Se instanciará un vector de enteros y otro de double en modo BENCHMARK para comprobar que se registran correctamente los tiempos de ejecución y que el vector es instanciado correctamente: inicializando sus estructuras de datos correctamente.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un vector de enteros y otros de double en tipo BENCHMARK 3. Retornar de la función *main*.
Resultado esperado	El sistema informará de la correcta inicialización de los miembros del vector y registrará los tiempos de ejecución según los requisitos.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-02, REQ-SI-04, REQ-SI-07, REQ-SI-09

PR-4: Instanciación de un vector con modo de reparto BENCHMARK

ID	PR-05
Título	Instanciación de vector con modo de reparto OPTIMIZADO.
Descripción	Se comprobará que le vector se instancia y que se leen los tiempos de ejecución correctamente.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un vector de enteros y otros de double con modo de reparto OPTIMIZED 3. Retornar de la función *main*.
Resultado esperado	El proceso inicializará correctamente las estructuras de datos del vector y recuperará los tiempos de ejecución del archivo.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-02, REQ-SI-07, REQ-SI-09, REQ-SI-20, REQ-SI-21, REQ-SI-22

PR-5: Instanciación de vector con modo de reparto OPTIMIZADO.

ID	PR-06
Título	Instanciación de vector con modo de reparto AD-HOC.
Descripción	Se instanciará un vector con modo de reparto AD-HOC y se comprobará que la inicialización de sus estructuras de datos se hace correctamente.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un vector de enteros y otros de double con modo de reparto AD-HOC. Los valores del vector de longitudes de los bloques serán: {1, INT_MAX, 10}; 3. Retornar de la función *main*.
Resultado esperado	El proceso inicializará correctamente las estructuras de datos del vector.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-02, REQ-SI-07, REQ-SI-09

PR-6: Instanciación de vector con modo de reparto AD-HOC

ID	PR-07
Título	Lectura de fichero binario con modo de reparto de BLOQUE
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo bloque.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto bloque. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos como estipula el reparto los elementos contenido en el archivo.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-7: Lectura de fichero binario con modo de reparto de BLOQUE

ID	PR-08
Título	Lectura de fichero binario, modo Round Robin, rodaja 1
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán, distribuidos como estipula el modo de reparto, los elementos leídos del archivo.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-8: Lectura de fichero binario, modo Round Robin, rodaja 1

ID	PR-09
Título	Lectura de fichero, modo Round Robin, rodaja igual a tamaño del vector
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos como estipula el reparto los elementos contenido en el archivo. El primer proceso contendrá todos los elementos.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-9: Lectura de fichero, modo Round Robin, rodaja igual a tamaño del vector

ID	PR-10
Título	Lectura de fichero, modo de reparto Round Robin, rodaja estándar.
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin con tamaño de rodaja mayor que 1 pero menor que el tamaño del vector.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos como estipula el reparto los elementos contenido en el archivo.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-10: Lectura de fichero, modo de reparto Round Robin, rodaja estándar

ID	PR-11
Título	Lectura de fichero binario con modo de reparto de BENCHMARK
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo BENCHMARK.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto BENCHMARK. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto BENCHMARK. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos como estipula el reparto los elementos contenido en el archivo. Además, los tiempos de ejecución de la prueba serán registrados.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10, REQ-SI-04,

PR-11: Lectura de fichero binario con modo de reparto de BENCHMARK

ID	PR-12
Título	Lectura de fichero binario con modo de reparto de OPTIMIZED
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo OPTIMIZED.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto OPTIMIZED. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto OPTIMIZED. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos como estipula el reparto los elementos contenido en el archivo. Se comprobará que la distribución de elementos ha sido hecha según los tiempos de ejecución registrados anteriormente.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10, REQ-SI-22

PR-12: Lectura de fichero binario con modo de reparto de OPTIMIZED

ID	PR-13
Título	Lectura de fichero binario con modo de reparto de AD-HOC
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo AD-HOC.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto AD-HOC. Las longitudes de los bloques serán las mismas que en la PR-06 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto AD-HOC. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Aplicar a los 3 vectores de enteros y a los 3 de double estas acciones, una a cada uno: a. Usando *read* con menos elementos que los del fichero b. Usando *read* con los mismos elementos que el fichero c. Usando *read* con más elementos que los que contiene el fichero. 6. Retornar de la función *main*.
Resultado esperado	Los vectores contendrán distribuidos, como estipula el modo de reparto, los elementos contenido en el archivo.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-13: Lectura de fichero binario con modo de reparto de AD-HOC

ID	PR-14
Título	Comprobar iteradores obtenidos con *begin* y *end*.
Descripción	Se utilizará un vector con modo reparto de BLOQUE para comprobar que los iteradores obtenido con estos métodos de clase cumplen los requisitos.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con al menos un elemento mediante la función ifstream::read 5. Invocar a los métodos begin y end del vector. 6. Retornar de la función *main*.
Resultado esperado	Los iteradores deberán apuntar a la primera posición (0) del vector y a la siguiente a la última.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-12

PR-14: Comprobar iteradores obtenidos con begin *y* end.

ID	PR-15
Título	Comprobación de operaciones de bucle con el iterador.
Descripción	Se comprobarán las operaciones típicas en el uso de un iterador en un bucle.

Pasos	1. Instanciar un objeto inicializador en la función *main*.
	2. Instanciar un DistributedVector de enteros con tipo de reparto bloque.
	3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros.
	4. Llenar el vector con al menos un elemento mediante la función ifstream::read
	5. Aplicar un bucle tal que:
	`for (auto ii = vector.begin(), ii != v.end(), ++ii){`
	`cout << *ii << endl;`
	`}`

Se accederá secuencialmente a todas las posiciones del iterador y se imprimirán por pantalla.	Se accederá secuencialmente a todas las posiciones del iterador y se imprimirán por pantalla.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-12, REQ-SI-13, REQ-SI-14

PR-15: Comprobación de operaciones de bucle con el iterador.

ID	PR-16
Título	Comprobación de operación preincremento e igualdad.
Descripción	Se comprobará que la prueba PR-15 funciona con el operador igualdad y preincremento.

Pasos	1. Instanciar un objeto inicializador en la función *main*.
	2. Instanciar un DistributedVector de enteros con tipo de reparto bloque.
	3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros.
	4. Llenar el vector con al menos un elemento mediante la función ifstream::read
	5. Aplicar un bucle tal que:
	`for (auto ii = vector.begin(), !(ii == v.end()), ii++){`
	`  cout << *ii << endl;`
	`}`

Resultado esperado	Se accederá secuencialmente a todas las posiciones del iterador y se imprimirán por pantalla.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-12, REQ-SI-13, REQ-SI-14

PR-16: Comprobación de operación preincremento e igualdad.

ID	PR-17
Título	Compatibilidad del iterador con la STL
Descripción	El iterador deberá ser compatible con los algoritmos de la STL con los que sea compatible el iterador de la clase vector.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con al menos un elemento mediante la función ifstream::read 5. Utilizar std::transform para incrementar en 1 el valor de cada elemento del vector.
Resultado esperado	Cada elemento del vector será incrementado sin necesitar cambios en la aplicación de la función transform de la STL.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-01, REQ-SI-12, REQ-SI-13, REQ-SI-14, REQ-SI-15

PR-17: Compatibilidad del iterador con la STL

ID	PR-18
Título	Comportamiento de la interfaz optimizada
Descripción	Se utilizará la interfaz optimizada de la biblioteca para incrementar un vector y luego imprimirlo por pantalla.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con al menos un elemento mediante la función ifstream::read 5. Utilizar la versión optimizada del operador de acceso para sumar 1 a cada elemento en el vector: (`vector[ii]++`).
Resultado esperado	El vector será incrementado en 1 utilizando la versión del operador de acceso que no usa *broadcast*.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-08

PR-18: Comportamiento de la interfaz optimizada

ID	PR-19
Título	Utilización de los métodos *set* y *get*
Descripción	Se comprobará que los métodos tienen el comportamiento requerido.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con al menos un elemento mediante la función ifstream::read 5. Utilizar *set* para aumentar en 1 el valor de todas las posiciones del vector, siendo éstas leídas con *get*.
Resultado esperado	El cambio en los elementos se realizará correctamente.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-08

PR-19: Utilización de los métodos set *y* get

ID	PR-20
Título	Algoritmo transform general
Descripción	Se probará que el algoritmo transform permite la copia de datos entre vectores y que se ejecuta correctamente.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un DistributedVector con reparto Round Robin de rodaja 1. 4. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 5. Llenar ambos vectores con 100 elementos mediante la función ifstream::read 6. Aplicar transform para incrementar en 1 los elementos del primer vector y guardarlos en el segundo. a. Con una función lambda b. Con un objeto-función c. Con un puntero a función
Resultado esperado	La transformación de los elementos y la copia de un vector a otro será realizada según lo esperable igual que si se usara std::transform.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-18

PR-20: Algoritmo transform general

ID	PR-21
Título	Algoritmo transform simple
Descripción	Se probará la ejecución del caso en que se aplique la transformación a todo un vector y se escriba sobre sí mismo.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con 100 elementos mediante la función ifstream::read 5. Aplicar transform para incrementar en 1 los elementos del vector y guardarlos en él mismo. a. Con una función lambda b. Con un objeto-función c. Con un puntero a función
Resultado esperado	El vector será incrementado y se informará mediante mensajes de depuración que se ha hecho mediante la ejecución asociada al caso simple.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-18

PR-21: Algoritmo transform simple

ID	PR-22
Título	Algoritmo reduce
Descripción	Se comprobará el funcionamiento del algoritmo *reduce*.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Llenar el vector con 100 elementos consecutivos empezando en 1 mediante la función ifstream::read 5. Aplicar *reduce* para sumar todos los elementos del vector a. Con una función lambda b. Con un objeto-función c. Con un puntero a función
Resultado esperado	La suma resultante será 5050.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-16, REQ-SI-17

PR-22: Algoritmo reduce

ID	PR-23
Título	Algoritmo reduce sin elementos
Descripción	Se comprobará el funcionamiento del algoritmo *reduce* cuando no hay elementos que sumar.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar un DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar un objeto de la clase ifstream que referencie a un archivo de enteros. 4. Aplicar *reduce* con dos iteradores iguales como argumentos. a. Con una función lambda b. Con un objeto-función c. Con un puntero a función
Resultado esperado	La suma arrojada será igual al valor elegido para *init*.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-16, REQ-SI-17

PR-23: Algoritmo reduce sin elementos

ID	PR-24
Título	Escritura de fichero binario con modo de reparto de BLOQUE
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo bloque.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto bloque. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto bloque. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. 7. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta la posición escrita.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-24: Escritura de fichero binario con modo de reparto de BLOQUE

ID	PR-25
Título	Escritura de fichero, reparto de Round Robin, tamaño de rodaja 1
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin con tamaño de rodaja 1.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin con tamaño de rodaja 1. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin con tamaño de rodaja 1. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. 7. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta esa posición.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-25: Escritura de fichero, reparto de Round Robin, tamaño de rodaja 1

ID	PR-26
Título	Escritura fichero, Round Robin, rodaja igual a tamaño del vector
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin.

Pasos	1. Instanciar un objeto inicializador en la función *main*.
	2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin con tamaño de rodaja igual al tamaño del vector.
	3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin con tamaño de rodaja igual al tamaño del vector.
	4. Instanciar dos objetos de la clase ifstream:
	  a. Uno conteniendo enteros
	  b. Otro contiendo doubles
	5. Llenar cada vector con 100 elementos mediante ifstream::read.
	6. Escribir los vectores en archivos diferentes con ifstream::write.
	  a. Indicando más elementos que la longitud de los vectores
	  b. Indicando los mismos elementos que la longitud del vector
	  c. Indicando menos elementos de los que tiene el vector.
	7. Retornar de la función *main*.

Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta la posición escrita.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-26: Escritura fichero, Round Robin, rodaja igual a tamaño del vector

ID	PR-27
Título	Escritura de fichero, reparto Round Robin, rodaja estándar.
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo Round Robin con un tamaño de rodaja mayor que 1 pero menor que el tamaño del vector.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto Round Robin. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto Round Robin. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. 7. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta esa posición.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-27: Escritura de fichero, reparto Round Robin, rodaja estándar.

ID	PR-28
Título	Escritura de fichero binario con modo de reparto de BENCHMARK
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo BENCHMARK.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto BENCHMARK. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto BENCHMARK. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta esa posición.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10, REQ-SI-04,

PR-28: Escritura de fichero binario con modo de reparto de BENCHMARK

ID	PR-29
Título	Escritura de fichero binario con modo de reparto de OPTIMIZED
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo OPTIMIZED.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto OPTIMIZED. 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto OPTIMIZED. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. 7. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta la posición escrita.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10, REQ-SI-22

PR-29: Escritura de fichero binario con modo de reparto de OPTIMIZED

ID	PR-30
Título	Escritura de fichero binario con modo de reparto de AD-HOC
Descripción	Se comprobará que la interfaz de lectura de dcpl::ifstream se comporta como se requiere en todos los casos contemplados para vectores en modo AD-HOC.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Instanciar 3 objetos DistributedVector de enteros con tipo de reparto AD-HOC. Las longitudes de los bloques serán las mismas que en la PR-06 3. Instanciar 3 objetos DistributedVector de double con tipo de reparto AD-HOC. 4. Instanciar dos objetos de la clase ifstream: a. Uno conteniendo enteros b. Otro contiendo doubles 5. Llenar cada vector con 100 elementos mediante ifstream::read. 6. Escribir los vectores en archivos diferentes con ifstream::write. a. Indicando más elementos que la longitud de los vectores b. Indicando los mismos elementos que la longitud del vector c. Indicando menos elementos de los que tiene el vector. 7. Retornar de la función *main*.
Resultado esperado	Los archivos de entrada y salida serán iguales, salvo en el caso de escribir menos elementos, entonces serán iguales hasta esa posición.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-11, REQ-SI-10

PR-30: Escritura de fichero binario con modo de reparto de AD-HOC

ID	PR-31
Título	Ejecución en modo optimizado con información no concluyente
Descripción	Se comprobará que la biblioteca es capaz de detectar que no se está ejecutando el programa en los mismos nodos en los que se ejecutó el *benchmark*, o que no hay archivo de datos en la ruta requerida.
Pasos	1. Ejecutar un programa con un vector en modo *benchmark*. 2. Cambiar el nombre de las máquinas o la configuración de ejecución para que sean otros nodos. 3. Ejecutar un programa con vectores en modo OPTIMIZED.
Resultado esperado	Se dará información al usuario sobre que no se ha encontrado información de rendimiento y se instanciará el vector en modo bloque.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-02, REQ-SI-05

PR-31: Ejecución en modo optimizado con información no concluyente

ID	PR-32
Título	Ejecución en modo optimizado en el mismo subconjunto de nodos
Descripción	Se comprobará que el programa es capaz de detectar que se está ejecutando en los mismos nodos, pero en diferente orden.
Pasos	1. Ejecutar un programa con un vector en modo *benchmark*. 2. Cambiar el nombre de las máquinas o la configuración de ejecución para que el orden de los nodos cambie. 3. Ejecutar un programa con vectores en modo OPTIMIZED.
Resultado esperado	El programa debería comportarse como si estuviéramos en el mismo conjunto de nodos que en la ejecución en la que se realizó el *benchmark*.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-02, REQ-SI-05

PR-32: Ejecución en modo optimizado en el mismo subconjunto de nodos

ID	PR-33
Título	Impresión con dcpl::cout
Descripción	Se comprobará que al utilizar la variable dcpl::cout, sólo el nodo 0 imprime los mensajes.
Pasos	1. Instanciar un objeto inicializador en la función *main*. 2. Utilizar std::cout para imprimir un mensaje: "A" 3. Utilizar dcpl::cout para imprimir un mensaje "B". 4. Retornar de la función *main*. 5. Ejecutar el programa en al menos 2 procesos.
Resultado esperado	"A" se imprimirá tantas veces como procesos hubieran ejecutado la sentencia, mientras que "B" sólo se imprimirá una vez.
Prueba superada	Sí
Requisitos relacionados	REQ-SI-23

PR-33: Impresión con dcpl::cout

7. Evaluación de rendimiento

En esta sección se van a describir las pruebas para evaluar el rendimiento de la biblioteca en diferentes situaciones. Se expondrán los resultados de las mismas y se dará una justificación razonada a los mismos. Las pruebas serán desarrolladas en 4 escenarios que permitirán evaluar diferentes facetas del rendimiento de la biblioteca:

1. Comparación con C++ secuencial en un solo nodo.
2. Comparación *transform* simple con *transform* general.
3. Comparativa de multicore y multinodo.
4. Comparativa de reparto de bloque y reparto optimizado.

Además, se desarrollarán una serie de pruebas en cada escenario con excepción del número 2, que tendrá una serie de pruebas propias. Cada una de estas pruebas tendrán un código asociado que se lista a continuación:

Prueba	Código	Procedimiento
***Transform* de todo el vector.**	TR	Se invocará a la función dcpl::transform para sumar una unidad a todo el vector
Reduce de todo el vector con operación suma	RD	Se sumarán todos los elementos del vector con la función dcpl::reduce.
Uso de operador corchete sobre todo el vector.	OP	Se utilizará el operador corchete para acceder a todas las posiciones del vector
Uso de get y set para todo el vector.	GS	Se usarán *get* y *set* para realizar la misma operación que en la prueba TR.
Lectura y escritura del vector en fichero a disco.	IO	Se evaluará el tiempo que se tarda en leer todo el vector de fichero y reescribirlo en otro fichero.

Tabla 12: Códigos asociados a las pruebas de rendimiento

7.1 Comparación con C++ secuencial en un solo nodo

En este escenario compararemos el *speedup* de una aplicación distribuida que utilice la biblioteca respecto a lo que tarda con un único proceso ejecutando la versión secuencial de los algoritmos, sólo con las herramientas de C++14. El computador utilizado para las pruebas es un nodo de tipo Compute 1 **[4.4]**.

Los resultados de las pruebas son los siguientes:

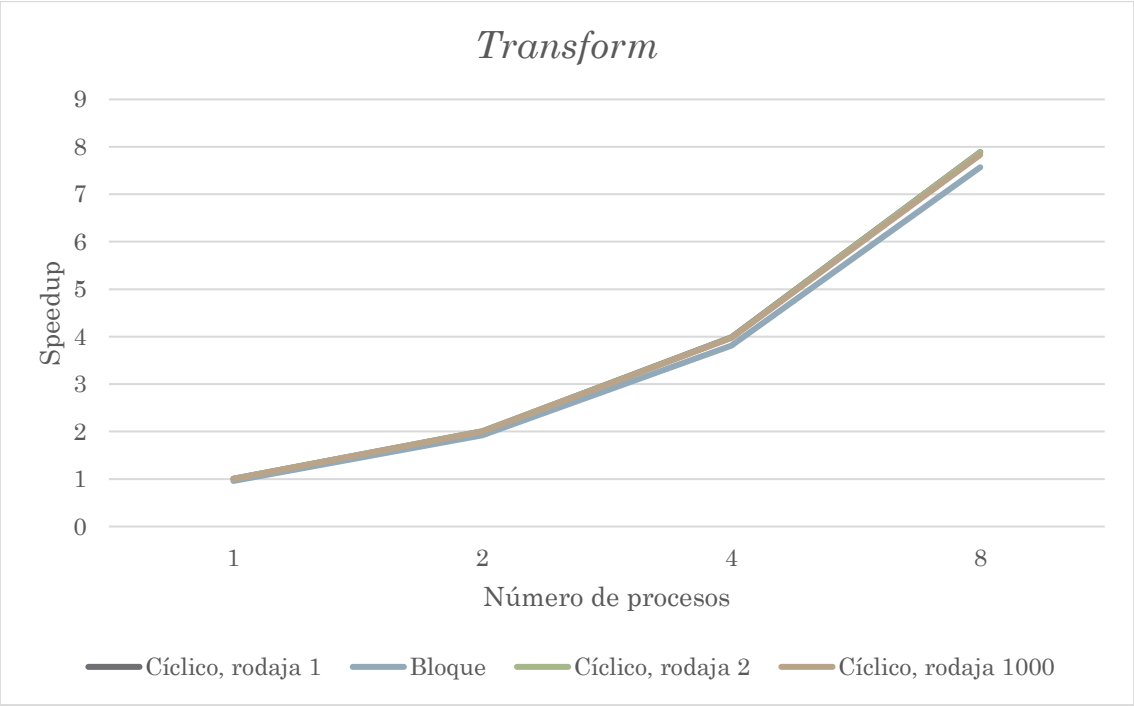


Gráfico 1: Speedup *para la prueba TR con diferentes modos de reparto*

En este caso, al no existir ninguna sobrecarga en la ejecución de la operación, dado que es el caso más simple de *transform*, simplemente cada proceso aplica la función *transform* de la STL a la totalidad de los datos que contiene en memoria. Esto hace que la comunicación entre procesos sea prácticamente nula y que, además, no sean necesarias comprobaciones sobre las posiciones del vector. Además, el tipo de reparto no influye en el rendimiento del algoritmo.

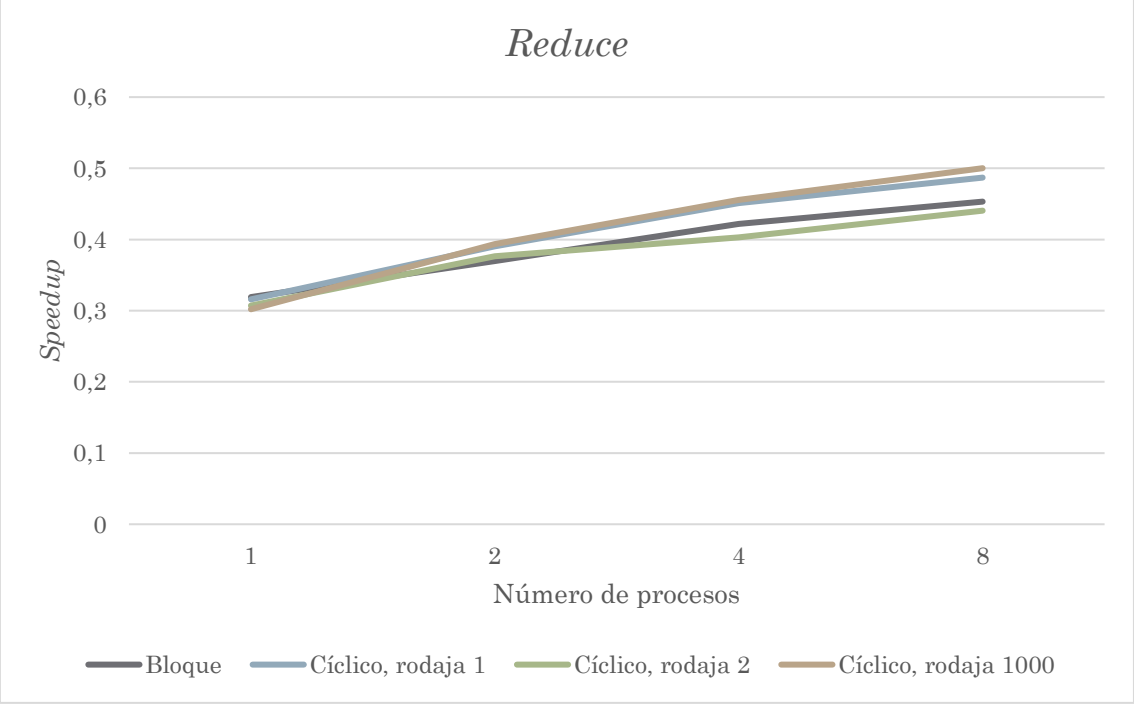


Gráfico 2: Speedup *para la prueba RD con diferentes modos de reparto*

En este caso, el *speedup*, aunque aumenta con el número de procesos involucrados, se sitúa siempre por debajo de 1 (es decir, nuestro algoritmo es más lento que la versión secuencial) y, además, el aumento del rendimiento con el número de procesos es cada vez menor según dichos procesos se ven incrementados. La explicación está en el algoritmo utilizado para calcular la suma parcial. Si el rango sobre el que se produce la operación *reduce* es de *n* elementos, cada proceso involucrado en el cálculo debe hacer *n* iteraciones en las que comprobará si aloja o no esa posición, como se describe en el Código 15. Esto hace que el tiempo mínimo de ejecución de la función sea el que tarde el proceso más lento en recorrer dicho bucle. Lo que provoca que la paralelización sólo se produzca en la ejecución de la función que represente el operador de reducción. Para que la paralelización fuera más efectiva se debería cambiar la filosofía actual de los algoritmos y calcular sólo la posición inicial y la final de la parte del vector local en la que el proceso debe ejecutar la función.

Además, se observan diferentes rendimientos según el modo de reparto escogido. Si bien los cambios no son muy significativos, se puede ver que el mejor modo de reparto es el cíclico con tamaño de rodaja 1.000 y que la diferencia se acentúa con el aumento del número de procesos. Esto es provocado porque en cada iteración del bucle donde los procesos calculan sus resultados parciales se invoca a la función *owner* **[OPERACIÓN 13: DV-03]**, la cual es más compleja y tiene mayor tiempo de ejecución en los modos de reparto de bloque. En los casos en que el proceso aloje la posición del vector, se invoca la función *global_to_local_pos* **[DV-04]**, que es más compleja cuando el modo de reparto es cíclico. Si esta función se invoca varias veces seguidas, el procesador puede realizar los cálculos más rápido, sin embargo, cuando se invoca pocas veces seguidas (con menor tamaño de rodaja), se penaliza más su ejecución.

Pero la función *owner* **[OPERACIÓN 13: DV-03]**, que calcula el proceso que aloja una posición dada, en el caso de que el modo de reparto sea cíclico con rodaja 1 se reduce a una operación módulo, lo que disminuye la sobrecarga de la llamada a la función en cada iteración del bucle.

Con estos datos, se concluye que el mejor modo de reparto para la operación *reduce* es cíclico con un tamaño de rodaja grande.

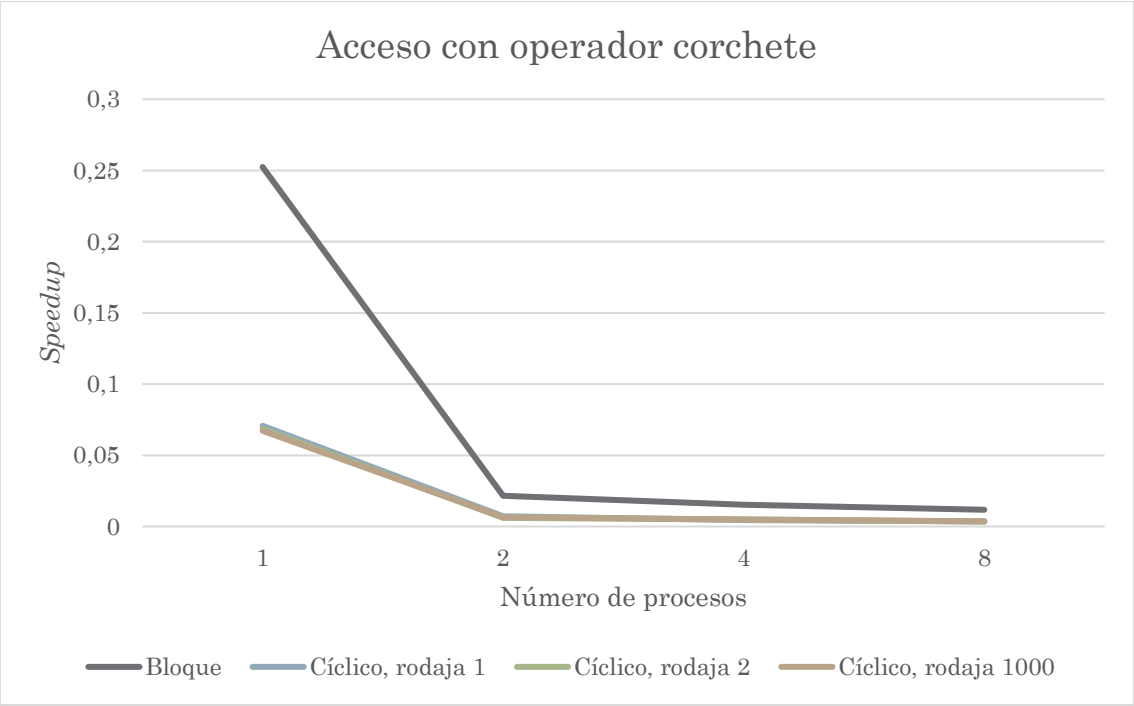


Gráfico 3: Speedup *para la prueba OP con diferentes modos de reparto*

En el caso de la operación de acceso mediante el operador corchete, el rendimiento es muy inferior al de la versión secuencial, sobre todo cuando el número de procesos es mayor que 1. Esto tiene una motivación sencilla: el operador corchete utiliza una operación de *broadcast*, que es muy costosa, y su coste aumenta con el número de procesos. El operador de acceso (corchetes) se pensó para operaciones puntuales, por ejemplo, fuera de bucles. Si se utiliza sobre un rango muy grande, se invierte mucho tiempo de ejecución en la comunicación de cada valor entre los procesos. Por otro lado, la diferencia entre los modos de reparto es casi imperceptible cuando el número de procesos es mayor que 2. Esto es porque el coste de realizar la operación de *broadcast* domina sobre el resto de operaciones. Finalmente, sí hay diferencias cuando el número de procesos es bajo. Esto es dado porque en estos casos la operación de *broadcast* tiene poco coste, esto hace que las diferencias de complejidad de la función *global_to_local_pos* **[DV-04]** tengan importancia en la ejecución del operador. Cuando el modo de reparto es de tipo bloque, el coste de dicha función es mucho menor que en modo de reparto cíclico. Lo que provoca que la diferencia en el caso de que sólo se ejecute un proceso sea notable.

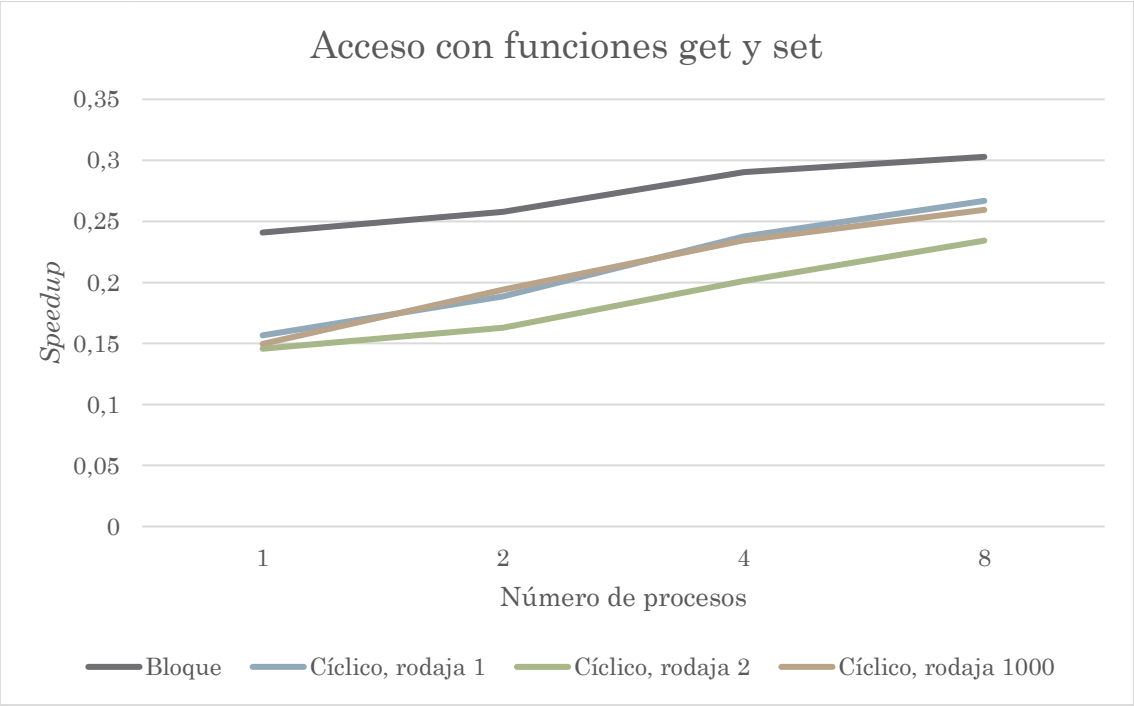


Gráfico 4: Speedup *para la prueba GS con diferentes modos de reparto*

En este caso el *Speedup* aumenta con el número de procesos ligeramente, y hay una diferencia notable entre diferentes modos de reparto, siendo el modo de reparto de bloque el que mejor rendimiento tiene. Esto es así porque en el escenario evaluado (en el que se hace *get* y *set* para acceder a una posición del vector y cambiar esa misma posición) la función ejecutada más veces es *owner* **[DV-03]**, esto, según la complejidad de la función, debería penalizar al modo de reparto en bloque, pero este efecto se ve paliado porque la complejidad de la función *global_to_local_pos* **[DV-04]** es mucho mayor en los casos en que el proceso que ejecute *get* y *set* aloje la posición. Esto hace que, en el caso de que sólo haya un proceso que ejecute la operación, todos ellos ejecuten, para cada posición 3 veces la función *owner* y 2 la función *global_to_local_pos*, esto hace que, según se aumente el número de procesos, cada proceso invoque menos veces la función *global_to_local_pos* respecto a *owner*. Lo que explica la mejoría de los repartos cíclicos respecto al reparto de bloque. El efecto del modo de reparto, sin embargo, no es determinante, dado que la diferencia en el *speedup* es de apenas un 7 %.

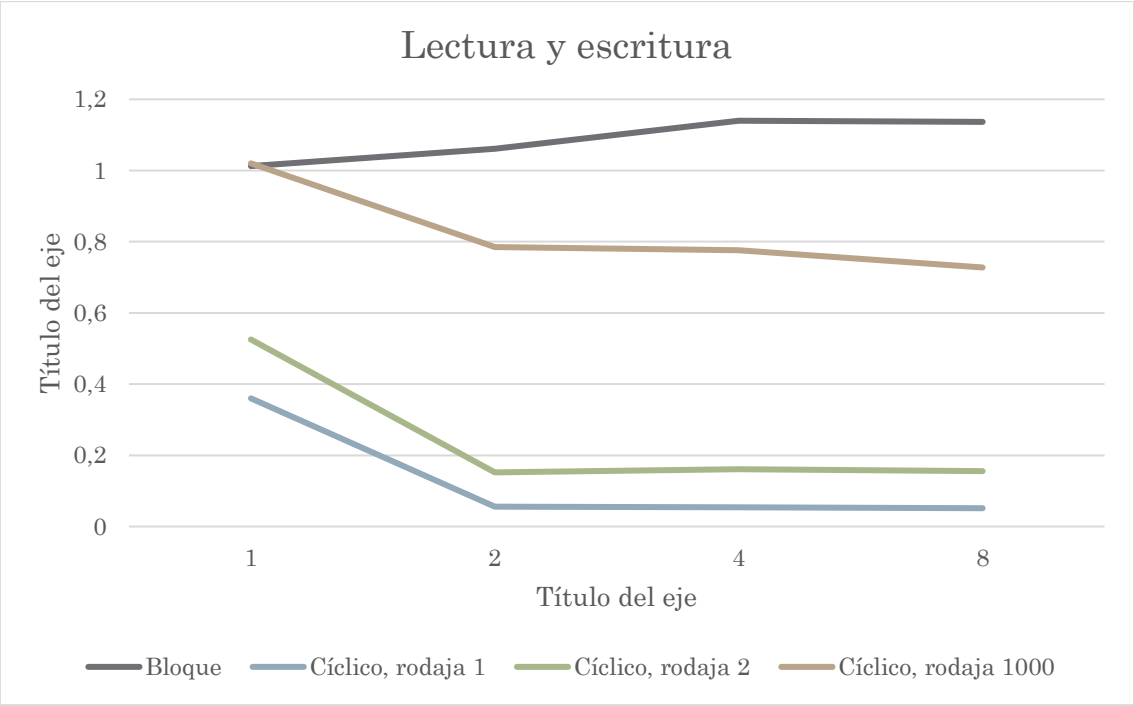


Gráfico 5: Speedup *para la prueba IO con diferentes modos de reparto*

En este caso, el modo de reparto es el factor que más afecta al rendimiento de la operación. Como se puede apreciar, el mejor de ellos es el modo de reparto en bloque, que, incluso, permite disminuir el tiempo de las operaciones respecto a la versión secuencial del programa. Le sigue el reparto cíclico con rodaja de 1.000 elementos y después, los repartos con rodaja de 2 y 1 elementos respectivamente. Esto es por lo siguiente: la lectura de varios bloques discontinuos del archivo que producen los modos de reparto cíclico provoca más llamadas al sistema y, sobre todo, peticiones de posiciones discontinuas hacia el sistema de ficheros. Esto provoca que el disco no atienda las misma eficientemente, en cambio, en un modo de reparto de bloque, cada proceso pide a disco una parte continua del fichero, en una sola llamada al sistema y de partes disjuntas. Por ello, el reparto de bloque permite aprovechar las características un sistema de ficheros distribuido con técnicas de caché o un sistema de RAID mucho mejor que el resto de repartos.

La conclusión de esta prueba es que, si se elige un modo de reparto cíclico, se debe tener en cuenta el impacto de la granularidad del mismo en el rendimiento de las operaciones de lectura y escritura a disco. Por ello, elegir una rodaja de un tamaño muy pequeño puede provocar múltiples llamadas a disco, bloqueándolo durante mucho tiempo y castigando su sistema de caché. En este caso, elidiendo una rodaja de tamaño 1.000 en un vector de 100 millones de elementos, se consigue una granularidad razonable, de unas 12.500 rodajas por proceso, en el caso de que se ejecuten 8 procesos. Lo que tiene un impacto limitado en el rendimiento, pero asumible.

7.2 Comparación *transform* simple con *transform* general

En la ejecución de la función *transform* se contemplan dos flujos de ejecución **[Caso de uso 12]**, las pruebas originales se evalúa el rendimiento de la versión simple, pero es necesario evaluar el rendimiento de la versión general del algoritmo. Por ello,

en este escenario se van a comparar ambas versiones. Además, se comparar en modo de reparto de bloque dado que se concluyó en **7.1** que el modo de reparto para el algoritmo era irrelevante. El computador utilizado para las pruebas es un nodo de tipo Compute 11 **[4.4 Entorno *hardware*]**.

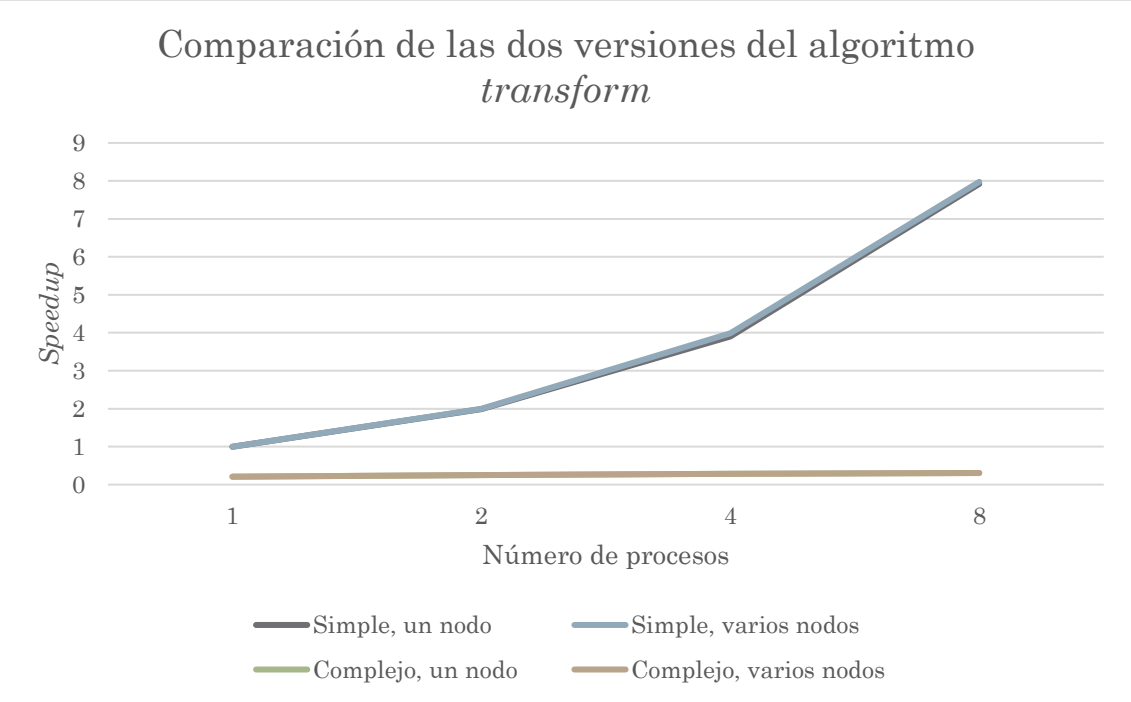


Gráfico 6: Comparación de transform simple y complejo

Como se puede ver, el *speedup* de la versión simple de *transform* es muy cercano al ideal, puesto que el paralelismo es total y apenas se requieren operaciones extra por parte de los procesos que en él intervienen.

Sin embargo, el *speedup* del caso general es menor que 1 en todo caso y parece que no se incrementa con el número de procesos. Sin embargo, en la siguiente figura en la que sólo aparece el modo complejo, se puede ver un pequeño incremento.

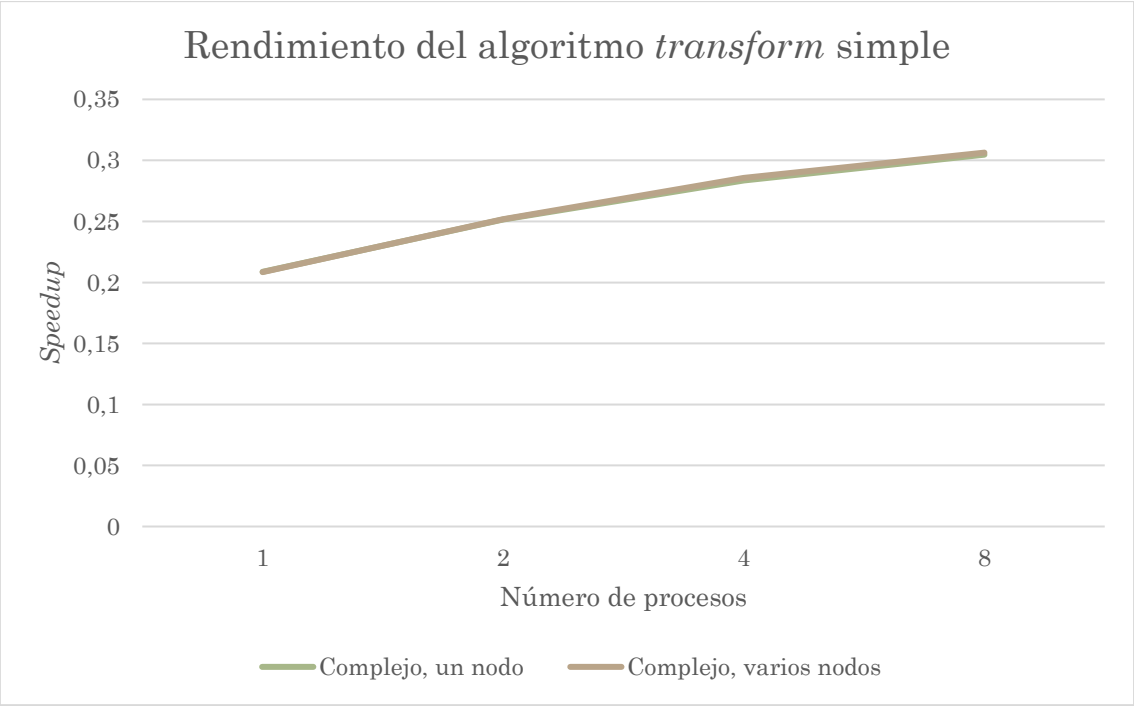


Gráfico 7: Algoritmos transform complejo en multinodo y en multicore

Esto indica que el rendimiento del algoritmo se comporta del mismo modo que reduce. Se paraleliza la ejecución del cálculo (en este caso la suma de 1 al elemento del vector), pero con cada proceso que se añade al sistema, se añade carga al mismo, pues todos deben recorrer el mismo bucle. De este modo, el rendimiento se ve penalizado.

Por otro lado, en ambos casos se ha comprobado el cambio que supone en el rendimiento la ejecución del programa en un nodo o en varios. Como se puede ver, la diferencia es imperceptible. Sin embargo, la versión ejecutada en varios nodos es marginalmente superior en rendimiento siempre. Esto es porque la sobrecarga producida por la comunicación es mínima en este caso, dado que, al producirse la transformación sobre el mismo vector, apenas hace falta comunicación entre procesos.

7.3 Comparativa de multicore y multinodo

Para realizar estas pruebas se han utilizado equipos con una CPU Intel Xeon E5-2603 v4 @ 1.70GHz – 12 núcleos perteneciente al clúster Tucan de la Universidad Carlos III de Madrid. (ver 4.4) Si han utilizado hasta 8 nodos con las mismas características para estas pruebas.

En este escenario el objetivo es comparar el rendimiento del programa según haya varios nodos interviniendo en clúster o todos los procesos se ejecuten en una misma máquina (siempre sin sobrepasar el número de núcleos de la misma). Esto no permitirá evaluar si hay una diferencia significativa en el rendimiento de las operaciones más habituales cuando se ejecutan en la misma máquina o en máquina separadas comunicadas por una red *ethernet*.

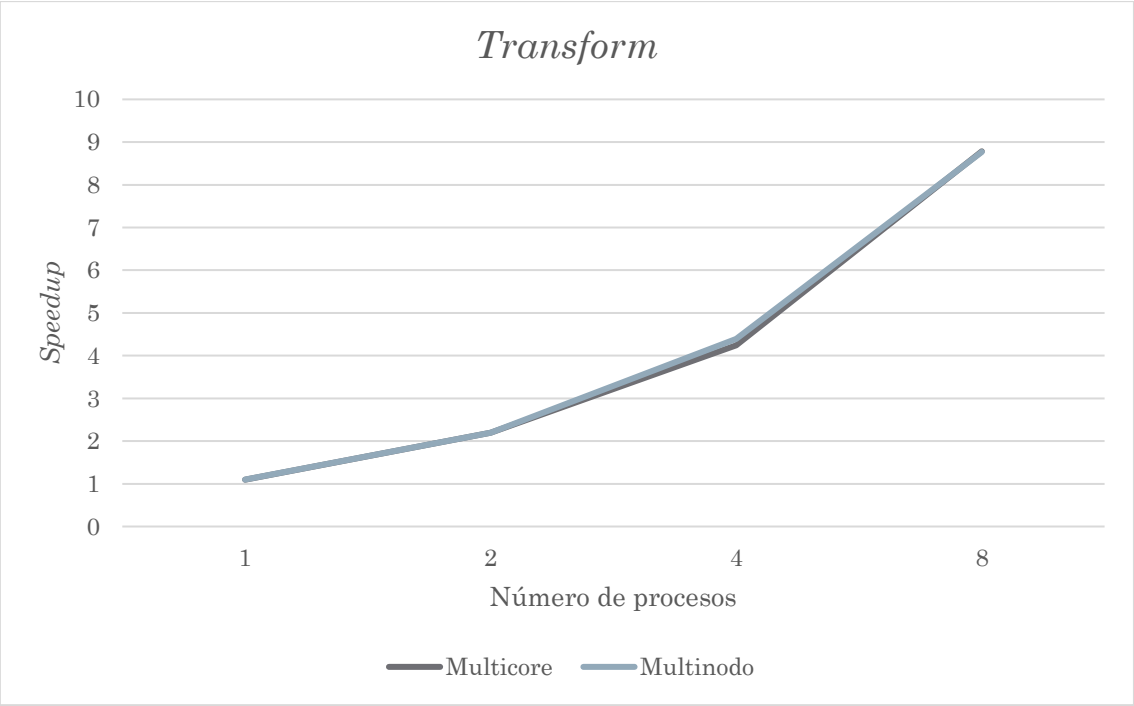


Gráfico 8: Prueba TR ejecutada en un nodo y en varios

Como se puede ver, en el caso de la operación *transform* en su versión simple no hay diferencia apreciable de rendimiento en ninguno de los casos. Esto se debe a que el uso de la red en este caso es prácticamente nulo y, además, la red de interconexión de las computadoras utilizadas para la prueba tiene un ancho de banda muy grande (10 Gbps) y una latencia mínima por distancia al estar en la misma sala.

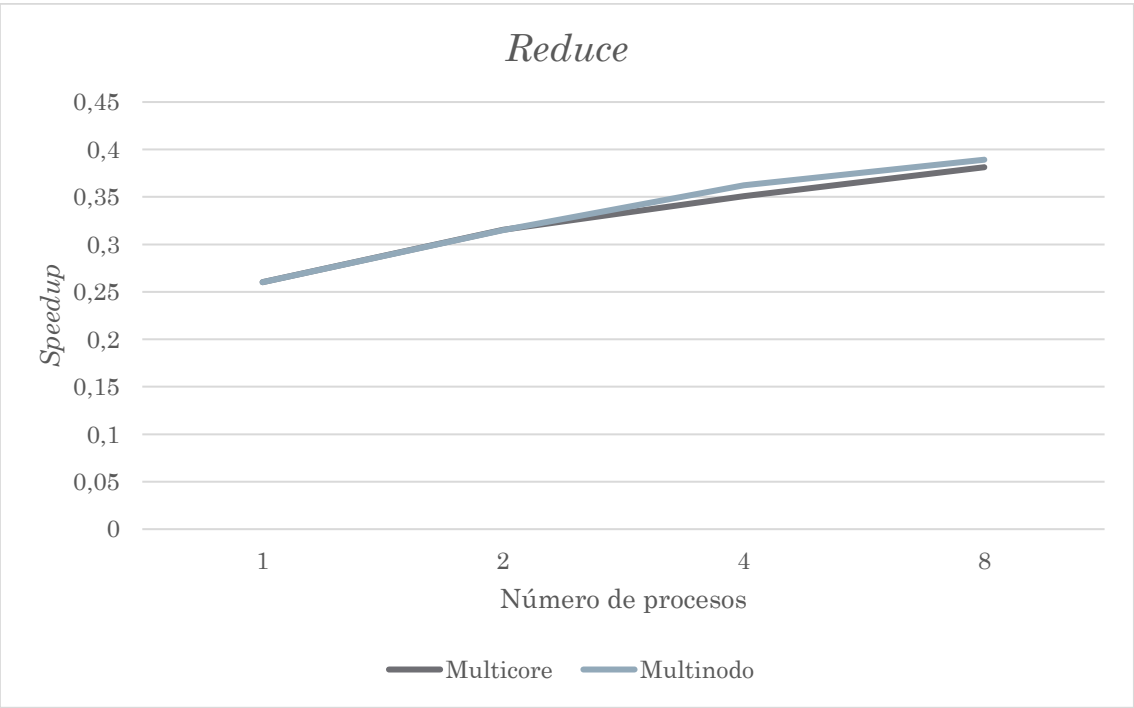


Gráfico 9: Prueba RD ejecutada en un nodo y en varios

Nos encontramos ante el mismo caso que en la prueba anterior, apenas hay ningún cambio de rendimiento, tanto si ejecuta en varios nodos como si lo hace en el mismo.

En este caso sí se hace uso de primitivas de comunicación (*broadcast* y creación de un comunicador). Pero ese uso apenas tiene efecto en el rendimiento total de la función.

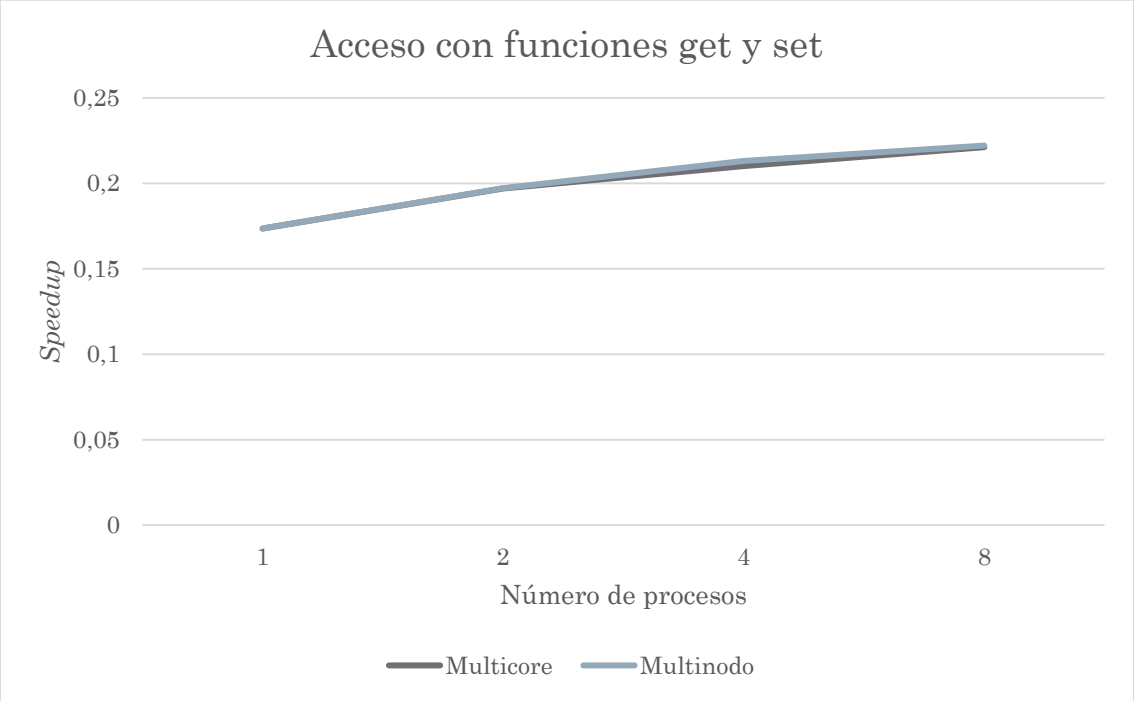


Gráfico 10: Prueba GS ejecutada en un nodo y en varios

En el rendimiento de los operadores *get* y *set* también se produce el mismo resultado, no hay apenas cambios entre una situación y otra.

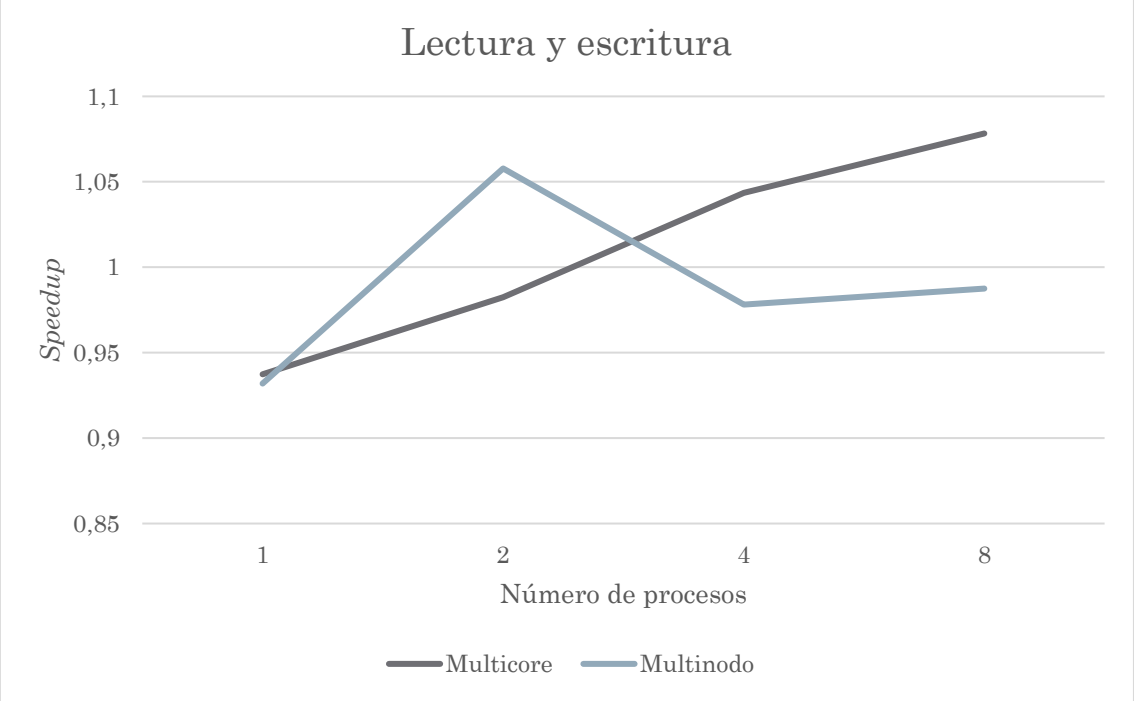


Gráfico 11: Prueba IO ejecutada en un nodo y en varios

Es en este caso en donde podemos ver mayor discrepancia de resultados. El sistema de ficheros donde se ha realizado esta prueba es un sistema de ficheros distribuido con tecnología NFS. Esto hace que todos los nodos lean del mismo disco físico. Cuando la ejecución se realiza en 2 nodos, el rendimiento es mayor que en dos procesos de la misma máquina, pero con 4 y 8 procesos, el rendimiento de la versión multicore es siempre mayor. Esto es así porque la gestión de peticiones que vienen del mismo computador para el sistema de ficheros es más fácil que la gestión de peticiones de varios. Si cada máquina contara con un sistema de ficheros propio donde estuvieran replicados los mismos datos, previsiblemente el rendimiento en multinodo sería mayor.

7.4 Comparativa de reparto de bloque y reparto optimizado

En el reparto optimizado se realiza un reparto de los elementos del vector inversamente proporcional al tiempo que tardaron en ejecutar un *benchmark* sobre el mismo código. Se supone que, si se ejecuta en nodos con rendimiento diferente, el modo de balanceo de carga provocará un aumento del rendimiento. Las pruebas se han ejecutado lanzando los procesos indicados en los resultados (dos, cuatro u ocho) en nodos con estas características:

- Nodos Compute 1:
  - Procesador: Intel Xeon E5405 @ 2 GHz
    8 núcleos
- Nodos compute 3:
  - Procesador: Intel Xeon E5405 @ 2GHz
    8 núcleos
- Nodos compute 6:
  - Procesador: Intel Xeon E5645 @ 2.4 GHz
    24 núcleos
- Nodos Compute 11:
  - Procesador: Intel Xeon E5-2603 v4 @ 1.70GHz – 12 núcleos

Los procesos han sido lanzados en los nodos en el orden mostrado, en el caso de la prueba con 8 procesos, se han lanzado dos procesos por nodo.

Los resultados de las pruebas se pueden ver en los gráficos 12, 13, 14 y 15.

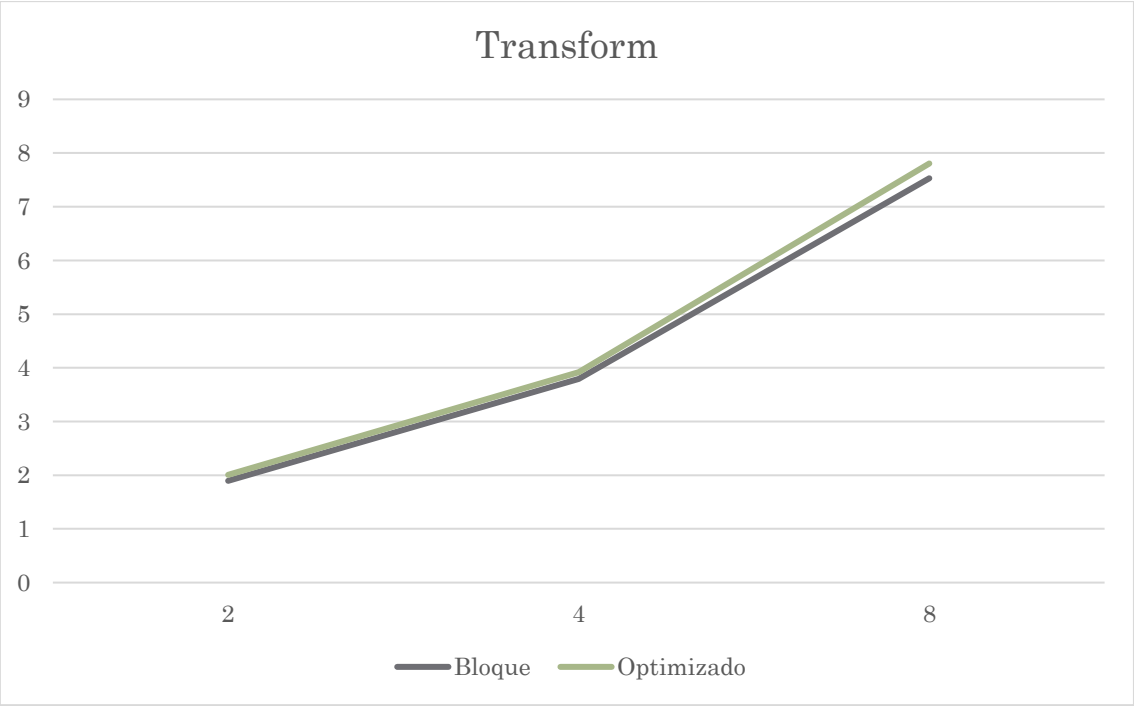


Gráfico 12: Prueba TR en modos de reparto de bloque y optimizado

Como se puede ver, en el caso del algoritmo transform simple, donde el paralelismo es casi total y apenas hay sobrecarga, hay una pequeña mejora de rendimiento en el modo optimizado. Los nodos nos estaban muy desequilibrados, dado que todos eran máquinas de altas prestaciones, pero pese a eso, sí se nota mejoría.

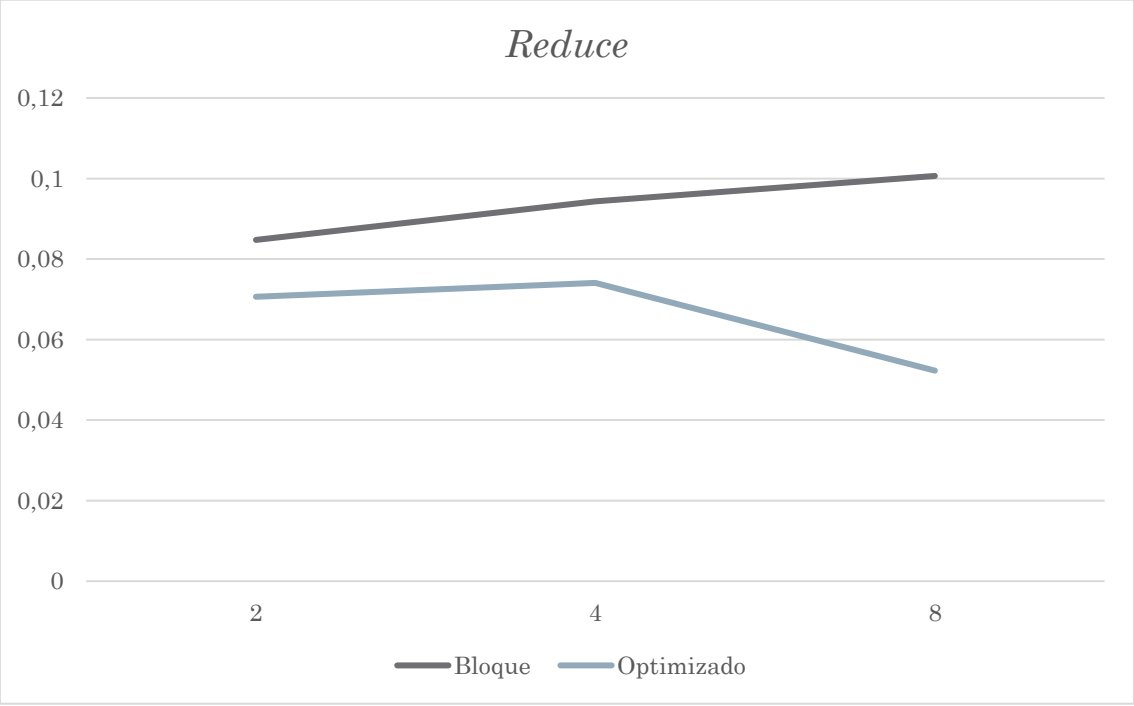


Gráfico 13: Prueba RD en modos de reparto de bloque y optimizado

En este caso, el modo de reparto optimizado tiene un rendimiento claramente peor que el modo de bloque. La explicación está, de nuevo, en las funciones *owner* y *global_to_local_pos*, en el caso del modo optimizado, son mucho más complejas que

en el modo de bloque y, además, tienen una complejidad lineal con el número de procesos, lo que explica, además, que la sobrecarga que producen sea mayor cuantos más procesos ejecuten la función. La mejora del algoritmo de balanceo de carga es insignificante comparada con la sobrecarga de las funciones.

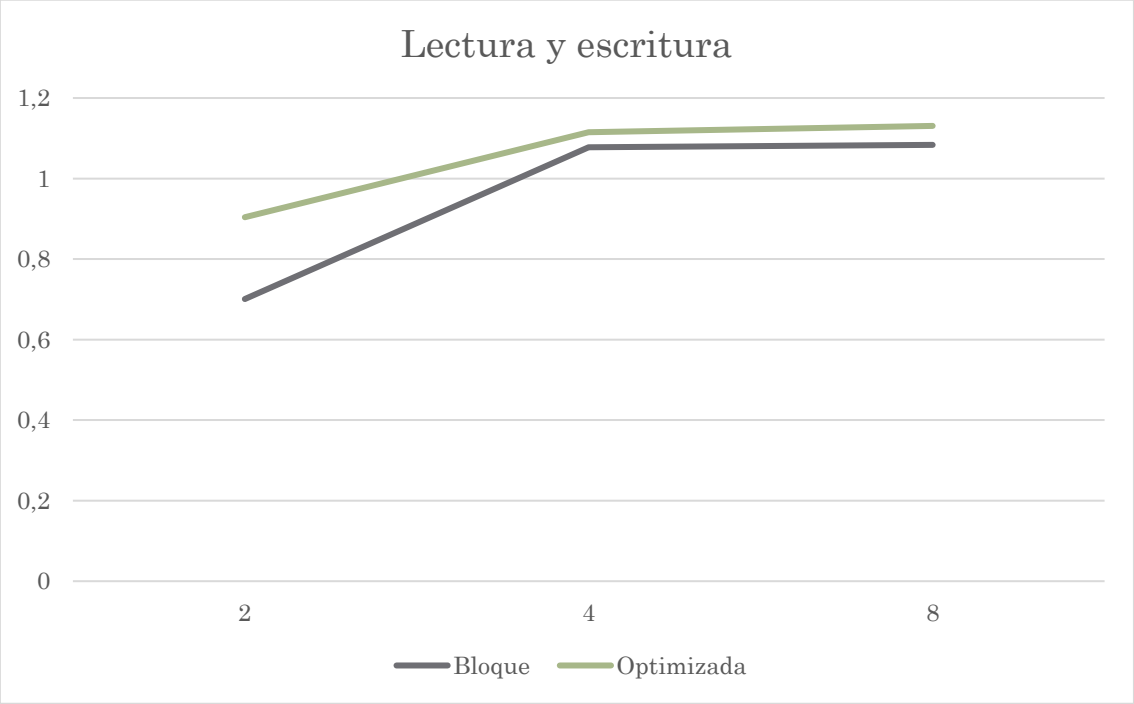


Gráfico 14: Prueba IO en modo de reparto de bloque y optimizado

En el caso de la entrada salida, la mejora es limitada, pero el modo de reparto optimizado es mejor que el reparto en bloque. Esto es porque para las operaciones de lectura y escritura no se utilizan las rutinas *owner* y *global_to_local_pos*, por ello, los nodos pueden beneficiarse del balanceo de carga. En operaciones de lectura y escritura de un sistema de ficheros distribuido, común cuando se trabaja con grupos de máquinas, el rendimiento del mismo puede varias para cada una. Dicho rendimiento depende, entre otras cosas, de la topología de la red y, por tanto, utilizar el balanceo de carga de la biblioteca para leer y escribir los ficheros permite que cada nodo realice la parte de la operación correspondiente al rendimiento que tiene en el uso del sistema de ficheros.

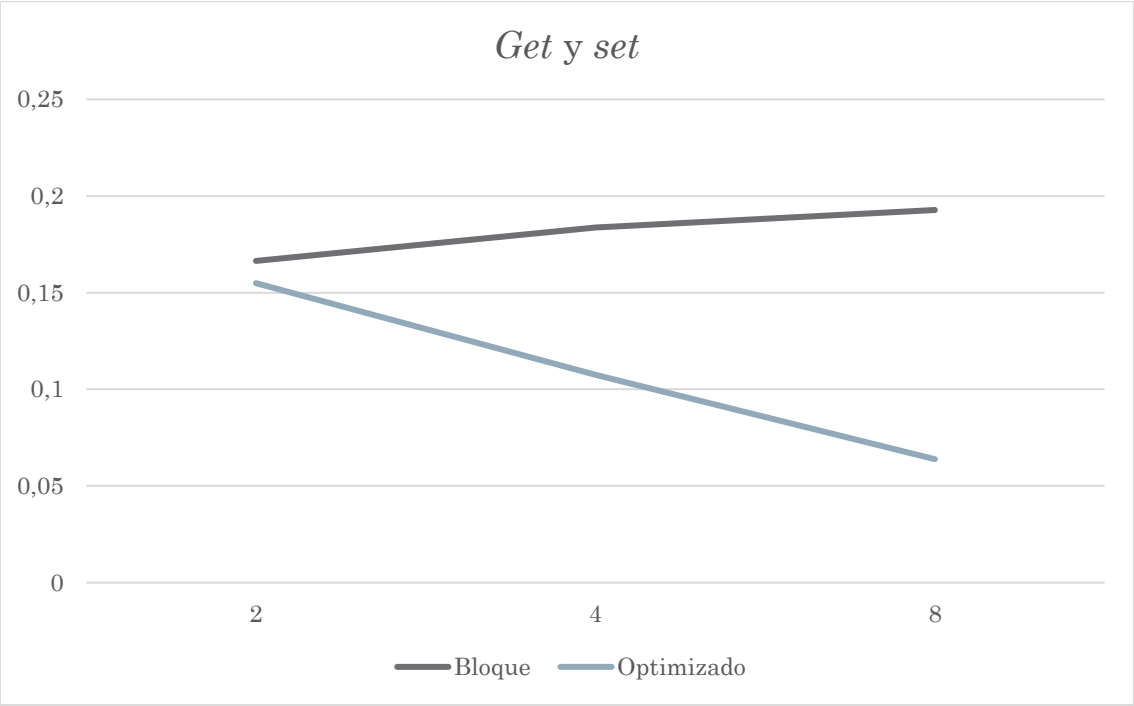


Gráfico 15: Prueba GS en modo de reparto de bloque y optimizado

En el uso de las funciones *get* y *set*, se aprecia un comportamiento similar al de la función *reduce*, esto es porque el flujo de ejecución es similar y, por tanto, se aprecia que el modo optimizado es peor cuantos más nodos hay en el sistema. Además, como en la operación realizada en esta prueba, el paralelismo aporta una mejora de rendimiento más pequeña que en el caso del algoritmo *reduce*, el efecto negativo de la complejidad del balanceo de carga enmascara la mejora de rendimiento proveniente del paralelismo.

8. Impacto socioeconómico

La comunidad de programadores en C++ tiene interés creciente por una solución como la que hemos desarrollado. Por otro lado, si se siguieran las líneas de trabajo futuro expuestas en la sección **9**, sería posible que el trabajo se incluyera en proyectos de mayor envergadura que pudieran conseguir financiación. Como el proyecto será distribuido como *software* libre, no existiría beneficio económico directo. Sin embargo, podría presentarse un impacto económico indirecto por proyectos en los que pudiera incluirse este desarrollo. La comunidad interesada en esto es grande debido a la extensión del uso de C++ y a la necesidad del uso de plataformas que utilizan memoria distribuida.

8.1 Presupuesto

En la presente sección se incluirán los equipos utilizados para el desarrollo de la aplicación y su evaluación, además de los costes del personal interviniente en el trabajo. A continuación, el detalle de los equipos utilizados:

Concepto	**Equipos de desarrollo**	**Equipo de pruebas**	**TOTAL**
Precio Unitario	€ 450,00	€ 12000,00	
Unidades	1	8	
Precio total	€ 450,00	€ 96000,00	
Periodo de amortización	36	36	
Amortización mensual	€ 12,50	€ 2666,67	
Amortizado en el proyecto	€ 37,50	€ 10666,67	**€ 10704,17**
Meses de proyecto	3	4	

Tabla 13: Coste de los equipos utilizados en el proyecto

Además, se incluirán los costes del personal del proyecto, incluyendo al autor, al tutor y a los profesores intervinientes que han colaborado, desglosando las retenciones del impuesto sobre la renta de las personas físicas, las retenciones de la seguridad social y el coste total del sueldo bruto.

	Francisco Rodríguez Melgar	**David Expósito Singh**	**José Daniel García Sánchez**	**TOTAL**
Rol	Autor	Experto en MPI y tutor	Experto en C++	—
Sueldo neto mensual	€ 2000,00	€ 3000,00	€ 3000,00	€ 8000,00
Retención IRPF anual	€ 5299,96	€ 11469,80	€ 1120,25	€ 17890,01
Retención SS anual	€ 1986,70	€ 2858,41	€ 2858,41	€ 7703,52
Sueldo bruto anual	€ 31286,66	€ 50328,21	€ 49978,67	€ 131593,54
Meses en el proyecto	3	3	3	
Retención por IRPF durante el proyecto	€ 1324,99	€ 2867,45	€ 280,06	€ 4472,50
Retención de la SS durante el proyecto	€ 496,68	€ 714,60	€ 714,60	€ 1925,88
Salario bruto durante el proyecto	€ 7821,67	€ 12582,05	€ 12494,67	**€ 32898,39**

Tabla 14: Coste del personal interviniente en el proyecto

Estos cálculos están basados en la normativa vigente a 31 de enero de 2017:

- Ley 35/2006 del 28 de noviembre.
- Ley 26/2014 del 27 de noviembre.
- Real Decreto-ley 9/2015 del 10 de julio.
- Real decreto 439/2007 del 30 de marzo.
- Real decreto 1003/2004 del 5 de septiembre.
- Real decreto 633/2015 del de 10 de julio.

Y en el supuesto de que el trabajador está en nómina un año, aunque sólo intervenga en este proyecto menos tiempo, para realizar los cálculos sobre un sueldo anual de 12 mensualidades. Se simula por tanto que interviene en el proyecto, pero realiza otras tareas para la organización.

Finalmente, se indica el coste del software utilizado:

Concepto	Microsoft Windows 10	Microsoft Office 2016	TOTAL
Precio Unitario	€ 60,00	€ 150,00	
Unidades	1	1	
Precio total	€ 60,00	€ 150,00	
Periodo de amortización	36	36	
Amortización mensual	€ 1,67	€ 4,17	
Amortizado en el proyecto	**€ 5,00**	**€ 12,50**	**€ 17,50**

Tabla 15: Coste del software *empleado en el proyecto*

El coste total del proyecto ascendería a:

Concepto	Coste
Software	€ 17,50
Equipos	€ 10704,17
Personal	€ 32898,39
Total	€ 43620,05
Porcentaje de riesgo	% 10,00
Total + riesgo	€ 47982,06
Margen de beneficio esperado	% 53,93
Ingreso bruto esperado	€ 73858,78
Beneficio neto esperado	€ 25876,72

Tabla 16: Costes totales del proyecto

Todos los costes estaban ya amortizados, ya sea por la universidad Carlos III de Madrid o por el autor del proyecto. Pero este presupuesto detallado indica la magnitud que podría llevar un proyecto así realizado por una empresa sin ninguna inversión previa al mismo. El margen de beneficio esperado es el beneficio medio del sector tecnológico en el primer cuatrimestre de 2017 **[35]**.

9. Conclusiones

En esta sección vamos a exponer las conclusiones técnicas de este desarrollo y un resumen de los resultados de las pruebas de rendimiento que se han llevado a cabo con él. Además, vamos a explicar cuál sería la línea de trabajo futuro que se debería llevar a partir de este desarrollo.

El objetivo principal de este software era proveer una interfaz igual que la secuencial y estándar de C++ para el manejo de vectores y algoritmos que utilicen vectores, pero que implementara una solución en memoria distribuida. Lo cual incluía permitir al usuario utilizar una versión de la clase vector que fuera almacenada en más de un proceso o máquina. Dicho objetivo ha sido cumplido, puesto que la interfaz de la biblioteca permite al usuario elegir el modo de reparto del vector y los datos serán almacenados en los nodos que intervengan en la ejecución sin más intervención suya a ese respecto.

Además, se quería proveer un mecanismo para que el usuario de la biblioteca fuera capaz de leer un archivo binario con una interfaz igual que la de C++ y utilizar los datos contenidos en él para llenar el vector. Esto se ha logrado casi por completo. La secuencia estándar para leer datos binarios e introducirlos en un vector en C++ es utilizar un objeto que representa el *stream* de *Bytes* y leer en la memoria reservada para el vector los *bytes* presentes en el fichero. Esto no se ha podido hacer exactamente así en la biblioteca porque no se trabaja con regiones de memoria, pero sí se ha podido ofrecer una interfaz compuesta de una clase que simboliza el stream y métodos *read* y *write* que cumplen la misma función que sus homólogos de la clase *ifstream* de C++.

El siguiente objetivo del trabajo era ofrecer al usuario una serie de algoritmos, representados por funciones, que permitieran operar con los vectores del mismo modo que con los vectores de la STL. La STL tiene una gran cantidad de algoritmos que funcionan con vectores, por ello, se eligió implementar dos de ellos: *transform* y *reduce*. Ambos algoritmos tienen una interfaz igual que sus homólogos de la STL y, además, permiten aprovechar toda la potencia de C++, que permite crear objetos invocables de diversos tipos. Esto permite que una persona que haya utilizado C++ asiduamente pueda entender perfectamente un código que utilice estos algoritmos.

Finalmente, el último de los objetivos principales de biblioteca es permitir al usuario la utilización de los vectores creados utilizando la misma sintaxis que un vector de C++. Esto incluye varias tareas: permitir utilizar un operador de acceso que se comporte igual que el de C++, permitir el uso de iteradores con las mismas cualidades que los del vector estándar de C++ y, además, hacer estos elementos compatibles con las funciones estándar de C++ que trabajan con ellos.

El operador de acceso de la biblioteca se comporta de tal modo que es capaz de devolver una referencia al elemento accedido del vector. Esto es útil para permitir operaciones compuestas con el operador o utilizar métodos que requieran una referencia como argumento. Además, permite mantener la coherencia en todos los procesos, lo que completa el objetivo de que el código sea igual que uno secuencial. Los iteradores implementados siguen todas las convenciones de C++, lo que permite su uso con funciones de C++ como *std::advance* o las propias de la STL, sin que el

usuario perciba diferencias. En suma, el objetivo de permitir un acceso al vector de una manera transparente para un programador en C++ ha sido cumplido.

Como objetivo secundario, se estableció que una ejecución en memoria distribuida fuera más rápida que la versión secuencial. Además, se ideó un algoritmo de balanceo de carga que permitiera aprovechar mejor la potencia de cómputo de clústeres no homogéneos. Si bien estos objetivos son importantes, cabe destacar que la ventaja principal de utilizar un contenedor distribuido como el propuesto es que permite procesar vectores que no podrían ser almacenados en la memoria principal de un único nodo.

Para evaluar si los objetivos de rendimiento se han cumplido se han realizado una serie de pruebas de rendimiento, a continuación, se lista un resumen del análisis de los resultados:

- El rendimiento de la función *transform* cuando corresponde a la utilización de la misma para aplicar una operación sobre todo el vector ha sido satisfactorio, siendo una operación simple, pero utilizada frecuentemente, es un resultado positivo.
- El rendimiento de la función reduce es malo, pero, desde el punto de vista teórico, lograr utilizar la operación MPI *Reduce* con cualquier función de C++ ha sido un paso importante en la adaptación de MPI a C++. Además, permite realizar este tipo de operaciones con vectores que no podrían ser alojados en un único ordenador.
- La optimización que MPICH presenta en operaciones de entrada y salida a disco ha permitido mejorar en muchos casos, aunque de manera modesta, el rendimiento de un programa secuencial en estas operaciones.
- Apenas se presenta sobrecarga en la ejecución de operaciones que impliquen trasmitir datos por red cuando se utilizan redes con alto ancho de banda gracias a las prestaciones de MPI.
- La sobrecarga del operador de acceso en su versión general es muy grande, de un modo que es prohibitivo para ser utilizado sobre muchas posiciones del vector. Sin embargo, cumple solventemente su función para operaciones puntuales, siendo totalmente transparente para el usuario.

9.1 Trabajo futuro

Es evidente que, si bien la interfaz es lo suficientemente transparente para el usuario que sólo conozca código en C++ y no sepa MPI, el objetivo del rendimiento no ha sido cubierto por el *software*, por ello, ésa sería la vía principal para futuros trabajos que tomen este como base, de este modo, el principal objetivo sería realizar una implementación más eficiente de los algoritmos ofrecidos por la biblioteca. En el caso de *reduce* dicho trabajo es más sencillo, pero en el caso de *transform*, un algoritmo que permitiera operaciones de *transform* entre vectores (aquéllas en las que *first* y *last* no referencien al mismo vector que *result*) sería previsiblemente bastante más complejo y por eso fue desestimado en el inicio del desarrollo de la biblioteca.

Por ello, las líneas principales que deberían seguir trabajos futuros serían

1. Aumentar el rendimiento de las funciones *transform* y *reduce*.
2. Hacer un análisis de los tipos de reparto presentes y mejorarlos.

3. Añadir más algoritmos distribuidos a la biblioteca.

Si bien la implementación realizada de los algoritmos distribuidos es intuitiva, tanto en cuanto utiliza la misma filosofía que la STL, haciendo una comprobación por elemento, no es eficiente, por lo que un trabajo futuro debería cambiar este enfoque hacia una filosofía de algoritmos pensados exclusivamente para memoria distribuida, sin tener que basarse en el comportamiento de los algoritmos de la STL.

10. Bibliografía

[1] STROUSTRUP, Bjarne. «The C++ programming language, 3ª ed.», 1997. ISBN: 0-201-88954-4

[2] What is Erlang?
https://www.erlang.org/
(Último acceso: 29 mayo 2017)

[3] COX, Russ. «Go, for Distributed Systems»
https://talks.golang.org/2013/distsys.slide#5
(Último acceso: 29 mayo 2017)

[4] ORACLE. «The History of Java Technology»,
http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html
(Último acceso: 29 mayo 2017)

[5] MICROSOFT DEVELOPER NETWORK. Porting Socket Applications to Winsock. Author.
https://msdn.microsoft.com/en-us/library/ms740096(VS.85).aspx
(Último acceso: 29 mayo 2017)

[6] W3C. URIs, URLs, and URNs: Clarifications and Recommendations 1.0.
https://www.w3.org/TR/uri-clarification/#uri-schemes
(Último acceso: 29 mayo 2017)

[7] IETF. Hypertext Transfer Protocol Version 2 (HTTP/2).
https://tools.ietf.org/html/rfc7540#section-9.1
(Último acceso: 29 mayo 2017)

[8] KREGER, Heather. «Web Services Conceptual Architecture», IBM Software Group.
http://www.csd.uoc.gr/~hy565/docs/pdfs/papers/wsca.pdf
(Último acceso: 29 mayo 2017)

[9] Seymour K., Nakada H., Matsuoka S., Dongarra J., Lee C., Casanova H. (2002) Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In: Parashar M. (eds) Grid Computing — GRID 2002. GRID 2002. Lecture Notes in Computer Science, vol 2536. Springer, Berlin, Heidelberg.
http://goo.gl/Fkt3A3
(Último acceso: 14 junio 2017)

[10] FOSTER, Ian. «Message Passing Interface», 1995.
http://www.mcs.anl.gov/~itf/dbpp/text/node95.html
(Último acceso: 30 mayo 2017)

[11] Meglicki, ZDZISLAW. «MPI IO», 2001, Indiana University.
http://beige.ucs.indiana.edu/B673/node179.html
(Último acceso: 30 mayo 2017)

[12] KIRK, Benjamin S., PETERSON, John et al. «libMesh: a C++ library for parallel adaptive mesh refinement/ coarsening simulations», 2006, Engineering With Computers 22(3-4):237-254.
DOI: 10.1007/s00366-006-0049-3.

[13] KAMBADUR, Prabhanjan, GREGOR, Douglas. «Modernizing the C++ Interface to MPI», 2006, Conference: Proceedings of the 13th European PVM/MPI Users' Group Meeting.
https://www.researchgate.net/publication/213882699_Modernizing_the_C_interface_to_MPI
(Último acceso: 30 mayo 2017)

[14] AN, Ping, JULA, Alin et al. «STAPL: An Adaptive, Generic Parallel C++ Library» 2003. DOI: 10.1007/3-540-35767-X_13.
https://link.springer.com/chapter/10.1007/3-540-35767-X_13
(Último acceso: 30 mayo 2017)

[15] BEAUMONT, David. «How to explain vertical and horizontal scaling in the cloud», 9 de abril de 2014, IBM.
https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/
(Último acceso: 2 junio 2017)

[16] SUTTER, Herb. «We have C++14!», 18 de agosto de 2014, Standard C++ Foundation.
https://isocpp.org/blog/2014/08/we-have-cpp14
(Último acceso: 2 junio 2017)

[17] «*Changelog* del compilador GCC», 21 de diciembre de 2016, Free Software Foundation.
https://gcc.gnu.org/gcc-6
(Último acceso: 2 junio 2017)

[18] VADHIYAR, Sathish S., DONGARRA, Jack J. «GrADSolve—a grid-based RPC system for parallel computing with application-level scheduling», 9 de septiembre de 2003, Journal of Parallel and Distributed Computing.

[19] CPLUSPLUS.COM. «Descripción del *header iterator*», autor.
http://www.cplusplus.com/reference/iterator/
(Último acceso: 2 junio 2017)

[20] IBM Corporation. «Overloading increment and decrement operators», autor.
https://www.ibm.com/support/knowledgecenter/en/SSB27U_6.4.0/com.ibm.zos.r12.cbclx01/cplr330.htm
(Último acceso: 7 junio 2017)

[21] RAFFO LECCA, Eduardo. «Programación genérica en C++, usando Metaprogramación», febrero de 2007, Industrial Data, p. 80-87. UNMSM.
http://sisbib.unmsm.edu.pe/bibvirtualdata/publicaciones/indata/vol10_n1/a1

2.pdf
(Último acceso: 8 junio 2017)

[22] BRACHA, Gilad. «Generics in the Java Programming Language», 2004, Oracle.
https://docs.oracle.com/javase/tutorial/java/generics/types.html
(Último acceso: 8 junio 2017)

[23] PACHECO, Peter S. «A User's Guide to MPI», 1998, p. 30-36, University of San Francisco.

[24] PLAUGER, P. J. «Standard C++ Library Reference», octubre de 2005, p. 249-274, Dinkumware, Ltd.
http://www.hpc.canterbury.ac.nz/UCSC%20userdocs/ForUCSCWebsite/C/AIX/standlib.pdf
(Último acceso: 13 junio 2017)

[25] CASS, Stephen. «The 2016 Top Programming Languages», 2016, IEEE Spectrum.
http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages
(Último acceso: 13 junio 2017)

[26] BARTLETT, Jonathan. «Introduction to metaprogramming», 2005, IBM Developer Works.
https://www.ibm.com/developerworks/library/l-metaprog1/
(Último acceso: 12 junio 2017)

[27] STEPANOV, Alexander. «STL and Its Design Principles», 2002, autor.
http://stepanovpapers.com/stl.pdf
(Último acceso: 12 junio 2017)

[28] BLOME, Michael, ROBETSON, Colin et al. «Lambda Expressions in C++», 2016, Microsoft.
https://docs.microsoft.com/es-es/cpp/cpp/lambda-expressions-in-cpp
(Último acceso: 13 junio 2017)

[29] BLOME, Michael, ROBERTSON, Colin et al. «friend (C++)», 2016, Microsoft.
https://docs.microsoft.com/es-es/cpp/cpp/friend-cpp
(Último acceso: 13 junio 2017)

[30] McCLEMENTS, Erik. «Performance Comparison of Open Source MPI Implementations», 2006, The University of Edinburgh.
https://static.epcc.ed.ac.uk/dissertations/hpc-msc/2005-2006/2688821-9h-dissertation1.1.pdf
(Último acceso: 18 junio 2017)

[31] «MPICH ABI Compatibility Initiative»
https://www.mpich.org/abi/
(Último acceso: 18 junio 2017)

[32] HAMMONS, Jack. «Bash on Ubuntu on Windows», 2017, MSDN.
https://msdn.microsoft.com/es-es/commandline/wsl/about
(Último acceso: 18 junio 2017)

[33] SUTTER, Herb, ALEXANDRESCU, Andrei. «C++ Coding Standards: Rules, Guidelines, and Best Practices (C++ in Depth)», 2004, Addison-Wesley. ISBN: 978-0-321-11358-0

[34] Argonne National Laboratory Group, «MPICH Copyright», 2002, autor.
https://svn.mcs.anl.gov/repos/mpi/mpich2/trunk/COPYRIGHT
(Último acceso: 18 junio 2017)

[35] CSI Market, «Technology Sector Profitability by quarter, Gross, Operating and Net Margin from 1 Q 2017», autor.
http://csimarket.com/Industry/industry_Profitability_Ratios.php?s=1000
(Último acceso: 18 junio 2017)